

# CANopen Manager-API

Software package for the development of  
CANopen Manager applications under Windows

---

Software Version 2.0

## **IXXAT**

### **Headquarter**

IXXAT Automation GmbH  
Leibnizstr. 15  
D-88250 Weingarten

Tel.: +49 (0)7 51 / 5 61 46-0  
Fax: +49 (0)7 51 / 5 61 46-29  
Internet: [www.ixxat.de](http://www.ixxat.de)  
e-Mail: [info@ixxat.de](mailto:info@ixxat.de)

### **US Sales Office**

IXXAT Inc.  
120 Bedford Center Road  
USA-Bedford, NH 03110

Phone: +1-603-471-0800  
Fax: +1-603-471-0880  
Internet: [www.ixxat.com](http://www.ixxat.com)  
e-Mail: [sales@ixxat.com](mailto:sales@ixxat.com)

## **Support**

In case of unsolvable problems with this product or other IXXAT products please contact IXXAT in written form by:

Fax: +49 (0)7 51 / 5 61 46-29  
e-Mail: [support@ixxat.de](mailto:support@ixxat.de)

## **Copyright**

Duplication (copying, printing, microfilm or other forms) and the electronic distribution of this document is only allowed with explicit permission of IXXAT Automation GmbH. IXXAT Automation GmbH reserves the right to change technical data without prior announcement. The general business conditions and the regulations of the license agreement do apply. All rights are reserved.

<b>1</b>	<b>INTRODUCTION .....</b>	<b>8</b>
1.1	Where to find What .....	9
1.2	Basic Specifications .....	9
1.3	Definitions, Acronyms, Abbreviations .....	10
1.4	Typographical Conventions .....	16
1.5	Support .....	16
1.6	Return of defect Hardware .....	16
<b>2</b>	<b>GETTING STARTED .....</b>	<b>17</b>
2.1	System Requirements .....	17
2.2	Supported CAN Boards .....	17
2.3	VCI .....	17
2.4	Installation .....	18
2.5	Flash Firmware .....	18
	2.5.1 VCI2 .....	18
	2.5.2 VCI3 .....	21
2.6	Becoming acquainted with the CANopen Manager .....	22
<b>3</b>	<b>OVERVIEW .....</b>	<b>25</b>
<b>4</b>	<b>TUTORIAL .....</b>	<b>26</b>
4.1	Setup of the Example Network .....	26
4.2	Initialization of the CANopen Manager .....	26
4.3	Configuration of the CANopen Manager via its local Object Dictionary .....	27
4.4	Generation of Process Images .....	31
4.5	CANopen Network Boot-up .....	34
4.6	Data Exchange via the Process Image .....	36
4.7	Auto Configuration Mode .....	39
4.8	Dynamic Generation of Object Dictionary Entries .....	40
<b>5</b>	<b>CANOPEN MANAGER FIRMWARE .....</b>	<b>42</b>
5.1	Overview .....	42
5.2	Services of the CANopen Manager .....	43
5.3	Boot-up Procedure .....	43
5.4	Network Management .....	44
5.5	RequestNMT Object .....	44

5.6	<b>Configuration Manager</b> .....	<b>45</b>
5.7	<b>Reset Configuration</b> .....	<b>45</b>
5.8	<b>Verify Configuration</b> .....	<b>45</b>
5.9	<b>Auto Configuration Mode</b> .....	<b>45</b>
5.10	<b>Initialization of the CANopen Manager</b> .....	<b>46</b>
5.11	<b>Object Dictionary default Values</b> .....	<b>47</b>
5.12	<b>Special Manufacturer-specific Object Dictionary Entries of the CANopen Manager</b> .....	<b>47</b>
5.13	<b>Configuration of the Run Time Behavior</b> .....	<b>50</b>
5.14	<b>Access to the local Object Dictionary</b> .....	<b>51</b>
5.14.1	CiA 301 specific object entries .....	51
5.14.2	CiA 302 specific object entries .....	51
5.15	<b>Dynamically created Object Dictionary Entries</b> .....	<b>54</b>
5.16	<b>Store/Restore</b> .....	<b>55</b>
5.17	<b>Handshaking</b> .....	<b>56</b>
<b>6</b>	<b>STRUCTURE OF THE PROCESS DATA INTERFACE</b> .....	<b>57</b>
6.1	<b>Process Data Interface</b> .....	<b>57</b>
6.1.1	Encoding rules.....	57
6.1.2	Data exchange between CMM-DLL and firmware.....	58
6.1.3	Overlaid Network Variables .....	58
6.1.4	Default values .....	59
6.1.5	RPDO no queue .....	60
6.1.6	TriggerTPDO queue.....	60
<b>7</b>	<b>DIAGNOSTICS DATA</b> .....	<b>61</b>
7.1	<b>Status Information of the CANopen Manager</b> .....	<b>61</b>
7.1.1	State of the CANopen Manager .....	62
7.1.2	Communication state of the CANopen Manager .....	64
7.1.3	Event Indication.....	65
7.1.4	Configuration of the CANopen Manager.....	67
7.2	<b>Slave Diagnostics</b> .....	<b>68</b>
7.2.1	Overview .....	68
7.2.2	Structure of the bit lists.....	69
7.2.3	Bit list assigned slaves .....	70
7.2.4	Bit list configured slaves.....	71
7.2.5	Bit list configuration error .....	71

7.2.6	Bit list operational slaves .....	74
7.2.7	Bit list stopped slaves .....	74
7.2.8	Bit list preoperational slaves .....	75
7.2.9	Bit list module internal errors .....	75
<b>7.3</b>	<b>Emergency Statistic and History .....</b>	<b>76</b>
7.3.1	Node Error Count .....	76
7.3.2	Error code-specific error counter .....	76
7.3.3	Emergency history .....	77
<b>7.4</b>	<b>Default Values .....</b>	<b>78</b>
<b>8</b>	<b>STATES OF THE CANOPEN MANAGER.....</b>	<b>79</b>
8.1.1	Initialization .....	80
8.1.2	Master Mode: Reset .....	80
8.1.3	Network Initialization .....	80
8.1.4	Auto Configuration .....	84
8.1.5	Network: Scanned .....	86
8.1.6	Network: Operational .....	88
8.1.7	Network: Stopped .....	89
8.1.8	Network: Pre-operational .....	90
8.1.9	Slave mode: Pre-operational .....	90
8.1.10	Slave mode: operational .....	91
8.1.11	Slave Mode: Stopped .....	91
8.1.12	Fatal Error .....	91
<b>8.2</b>	<b>Description of the State Transitions .....</b>	<b>92</b>
<b>9</b>	<b>CANOPEN MANAGER API – FUNCTIONALITY SUMMARY .....</b>	<b>96</b>
<b>10</b>	<b>CANOPEN MANAGER API DLL.....</b>	<b>98</b>
<b>10.1</b>	<b>Function Categories .....</b>	<b>98</b>
10.1.1	Basic functions .....	99
10.1.2	General functions .....	99
10.1.3	Functions for network management .....	100
10.1.4	Object dictionary and SDO-related functions .....	100
10.1.5	Process image-related functions .....	101
<b>11</b>	<b>INDIVIDUAL FUNCTIONS OF THE API-DLL .....</b>	<b>102</b>
<b>11.1</b>	<b>Basic Functions.....</b>	<b>103</b>
11.1.1	CMM_InitBoard .....	103
11.1.2	CMM_ReleaseBoard .....	105

11.1.3	CMM_GetBoardInfo.....	106
11.1.4	CMM_InitFirmware.....	107
11.1.5	CMM_DefineCallbacks.....	109
11.1.6	tCMM_CALLBACK.....	110
11.1.7	CMM_ResetDLL.....	111
11.1.8	CMM_SetCommTimeout.....	112
11.1.9	CMM_SetInspeInterval.....	113
11.1.10	CMM_DefineMsgProclmg.....	114
11.1.11	CMM_DefineMsgMaster.....	115
11.1.12	CMM_DefineMsgSlaves.....	116
11.1.13	CMM_DefineMsgEvent.....	117
11.1.14	CMM_DefineMsgEmergency.....	118
<b>11.2</b>	<b>General Functions.....</b>	<b>119</b>
11.2.1	CMM_GetMasterStat.....	119
11.2.2	CMM_GetSlavesStat.....	120
11.2.3	CMM_GetEvent.....	121
11.2.4	CMM_GetEmergencyObj.....	125
11.2.5	CMM_SendEmergencyObj.....	126
11.2.6	CMM_HandShake.....	127
<b>11.3</b>	<b>Functions for Network Management.....</b>	<b>128</b>
11.3.1	CMM_StartBootupProc.....	128
11.3.2	CMM_StartAutoConfig.....	129
11.3.3	CMM_StartNode.....	130
11.3.4	CMM_StopNode.....	131
11.3.5	CMM_EnterPreOp.....	132
11.3.6	CMM_ResetComm.....	133
11.3.7	CMM_ResetNode.....	134
<b>11.4</b>	<b>Object Dictionary and SDO related Functions.....</b>	<b>135</b>
11.4.1	CMM_CreateODentry.....	135
11.4.2	CMM_ReadSDO.....	137
11.4.3	CMM_WriteSDO.....	139
11.4.4	CMM_ReadLocSDO.....	140
11.4.5	CMM_WriteLocSDO.....	141
11.4.6	CMM_ImportCDC.....	142
<b>11.5</b>	<b>Process image-related functions.....</b>	<b>143</b>
11.5.1	CMM_FormPILUT.....	143
11.5.2	CMM_GetPIdescr.....	144

11.5.3	CMM_GetPI .....	146
11.5.4	CMM_GetPIentry .....	147
11.5.5	CMM_GetPIIvalue .....	148
11.5.6	CMM_PutPIO .....	149
11.5.7	CMM_PutPIOentry .....	150
11.5.8	CMM_PutPIOvalue .....	151
11.5.9	CMM_GetPIIRPDOno .....	152
11.5.10	CMM_TriggerPIOoffset .....	153
<b>APPENDIX – SCOPE OF DELIVERY .....</b>		<b>154</b>

# 1 Introduction

The CANopen Manager Application Programming Interface (API) is a program library for connecting a PC application to a CANopen network.

The available functionality includes a CANopen Master with interfaces for control, diagnostics data, data exchange by means of client SDOs, as well as a process image. In addition the CANopen Manager implements the functionality of a CANopen configuration manager.

The functionality of the CANopen Manager API adheres to the CANopen Application Layer and Communication Profile (CiA 301, [1]) and the CANopen Framework for CANopen Managers and Programmable CANopen Devices (CiA 302, [2]).

The software supports Microsoft Windows 2000, Windows XP, Windows XP64, and Windows Vista.

The user has to be familiar with the basic mechanisms and terms of CANopen. Further information can be obtained from the related specifications, which are available from CiA ([www.can-cia.org](http://www.can-cia.org)).

An introduction to CANopen is also given in the book *Controller-Area-Network, Basics, Protocols, Chips and Applications* by K. Etschberger, 2001, IXXAT Press, ISBN 3-00-007376-0.

Additional up-to-date information on the software not contained in this manual may available in the form of **ReadMe** files on the product CD-ROM.



### 1.1 Where to find What

This manual contains a description of the CANopen Manager and of all functions and data structures provided by the CANopen Manager API.

A description of how to get started with the CANopen Manager API can be found in chapter 2, *Getting Started*.

Chapter 3, *Overview* provides a general introduction to the software. A step-by-step introduction to the CANopen Manager API is found in chapter 4, *Tutorial*. The firmware of the CANopen Manager API is described in chapter 5, *CANopen Manager Firmware*. The structure of the data interfaces exported by the CANopen Manager API DLL to client applications is discussed in chapter 6, *Structure of the Process Data Interface* and chapter 7, *Diagnostics Data*. Chapter 8, *States of the CANopen Manager* describes the various states of the CANopen Manager and their transitions.

An introduction to the functions of the API can be found in chapter 10, *CANopen Manager API DLL*, whereas chapter 11, *Individual Functions of the API-DLL* contains a detailed description of the API.

### 1.2 Basic Specifications

- [1] CiA 301 CANopen Application Layer and Communication Profile V4.02, February 2002
- [2] CiA 302 CANopen Framework for CANopen Managers and Programmable CANopen Devices V3.3.0, October 2003
- [3] CiA 306 Electronic Datasheet Specification for CANopen, V1.3, January 2005
- [4] CiA 401 CANopen Device Profile for Generic I/O Module, V2.1, May 2002
- [5] CiA 405 Interface and Device Profile for IEC 61131-3 Programmable Devices, V2.0, May 2002

### 1.3 Definitions, Acronyms, Abbreviations

#### API

Application Programming Interface

#### Boot-up message

The boot-up message is a one byte CAN object that is transmitted by a CANopen slave on transition from **Initialisation** to **Pre-operational** state. See also NMT.

#### Boot-up procedure

The boot-up procedure is carried out according to CiA 302 [2] and is used for initialization of the network.

#### Boot slave process

Boot slave process of the CiA 302 [2]. During the boot slave process the identity of a slave module is determined and the slave module is configured.

#### Boot time

Additional object entry of CiA 302: [1F89]. An error is indicated if not all mandatory slaves have booted after the expiration of this configurable time.

#### CAN-ID

The CAN message identifier is used to uniquely flag a CAN message, and also defines the priority of a message. The highest priority CAN-ID 0 (11-bit identifier) is reserved for network management services (→ NMT).

#### CANopen manager

In addition to the standard CANopen functionality, a CANopen manager comprises the NMT master and at least one of the following functionalities:

**SDO manager** or → **Configuration manager**

#### CiA

CAN in Automation e.V.: Vendor and user organization.  
See also [www.can-cia.org](http://www.can-cia.org)

#### Client SDO/CSDO

A client SDO is the initiator of an SDO transmission. It has access to the object dictionary entries of a **SDO server**. → SDO

### **COB: Communication object**

A COB is a CAN message that is transmitted in the CAN network. Data are transported with a COB.

### **Communication parameters**

The attributes of a → PDO are described by its communication parameters. The attributes include → **transmission type**, → **inhibit time**, and the → COB-ID.

### **Configuration manager**

A configuration manager carries out configuration of the individual slave modules as part of the boot slave process.

### **Configure slave**

Additional object entry of CiA 302: [1F25]. By writing the signature **conf** to the corresponding sub index the boot slave process for a module can be requested.

### **COB-ID**

The COB-ID contains the CAN-ID (message identifier) plus additional configuration information.

### **Concise DCF**

Additional object entry of CiA 302: [1F22]. The configuration data of the individual slave modules are stored in this object entry. The slave modules are configured with these configuration data by the configuration manager during booting. To reduce memory requirements, a concise DCF only contains those object entries that differ from the default values.

### **DCF**

Device configuration file according to CiA 306 [3]

### **Error control service**

Cyclic monitoring of a node. Node monitoring can be implemented either via → **node guarding** requests or via → **heartbeat**.

## Introduction

---

### Error control event

The error control service detected an error when monitoring a node.

### Expected configuration date

Additional object entry of CiA 302: [1F26]. → Expected configuration time.

### Expected configuration time

Additional object entry of CiA 302: [1F27]. These object entries specify the date and time for identification of the configuration of a slave module and serve to reduce the time required for the boot slave process. During the boot slave process these values are compared with the values of the **verify configuration** object [1020] of the slave module, if this is supported by the module.

### Heartbeat

Node monitoring mechanism that was introduced with CANopen V4. The heartbeat mechanism is based on a producer – consumer model in which CANopen modules cyclically transmit their current NMT state. Contrary to → **node guarding**, this mechanism does not require any CAN remote frame requests.

### Identity objects

Additional object entries of CiA 302: [1F84]..[1F88]. These objects describe the expected device type and identity of the slave modules:

- Device-ID [1F84]
- Vendor-ID [1F85]
- Product code [1F86]
- Revision number [1F87]
- Serial number [1F88]

### Mandatory/optional

To categorize objects and services the CANopen specification uses the terms **mandatory**, **optional**, and **conditional**. The implementation of mandatory objects is required by the related specifications.

### NMM

Network management master

### NMS

Network management slave

### **NMT: Network management**

Service element of the application layers in the CAN reference model, which comprises the network-wide process synchronization and error control. CANopen has four main states: **Initialization**, **Pre-operational**, **Operational**, and **Stopped**. The status transition of a CANopen node is requested with NMT commands. The network management is based on a master slave structure.

### **NMTStartup**

Additional object entry of CiA 302: [1F80]. This object entry is used to configure the start-up behavior of a module.

### **Node guarding**

Cyclic guarding of a node. The NMT master cyclically transmits node guarding requests (remote frame request) to each slave, which individually reply with their node status. See also → **Heartbeat**.

### **Node-ID**

A device in the CAN-network is uniquely identified by its node-ID (between 1 and 127). This node-ID is used by → **predefined connection set** for the pre-defined identifier allocation. In a CANopen network each node-ID may only be used once. The CANopen Manager is a regular network node and thus also has a node-ID.

### **OD: Object dictionary**

Device internal logical addressing scheme to reference both configuration and application parameters.

The object dictionary is a data structure via which all objects of a CANopen device can be addressed. The object dictionary is subdivided into an area with general information on the device like identification information and communication parameters and an area that describes application specific device functionality. The data in the object dictionary are addressed via an index and a sub index. Via the entries (objects) of the object dictionary, the application objects of a device such as input and output signals, device parameters, function or network variables are made accessible in standardized form via the network. The object dictionary forms the interface between network and application process.

### **PDO: Process data object**

PDOs represent the actual means of transport for the transmission of process data. A PDO is transmitted by a **PDO producer** and can be received by one or more **PDO consumers**. The process data transmitted by a producer in a PDO can

## Introduction

---

contain a maximum of 8 bytes. A PDO is transmitted without acknowledgement and requires a unique → **CAN-ID** allocated to the PDO. The PDO producer manages the configuration information required by the PDOs in the form of so-called TPDO data structures, the data to be received by a PDO consumer are managed by so-called RPDO data structures. The communication-specific parameters specify the mode of the PDO and CAN-ID to be used. The data content of the transmitted data is specified in PDO mapping structures.

### **PI**

Process Image. Process data which can be read and written by the client application. Divided into → **PI input** and → **PI output**.

### **PI input**

Input data of the process image, received in → RPDO.

### **PI output**

Output data of the process image, transmitted via → TPDO.

### **Predefined connection set**

Preset allocation of the → COB-ID based on the → node-ID and on a 4-bit function code. The 127 nodes are differentiated via the least significant seven bits of the identifier. For the following communication objects, the predefined connection set predefines the COB-ID: Node guarding, heartbeat, emergency message, SYNC object, time stamp, first server SDO, RPDO1 to RPDO4, and TPDO1 to TPDO4.

### **RequestNMT**

Additional object entry of CiA 302: [1F82]. With this object entry both the state of the individual modules and the execution of NMT commands can be requested.

### **RPDO: Receive PDO**

→ PDO

### **SDO: Service data object**

An SDO is a CAN communication object which is used for initialization and parameterization of CANopen devices or for transmission of longer data records. SDOs are used for read or write access to the entries in the object dictionary of a device. A particular entry is addressed by its index and sub index.

### **SDO timeout**

An SDO request has to be responded to within the timeout time, otherwise the SDO will be aborted.

### **Server SDO/SSDO**

Each device must support at least one server SDO and thus allow access to the entries in its object dictionary. The specification of an SDO server object requires the definition of one CAN-identifier per transmission direction (acknowledged service), and specification of the corresponding client or server node if dynamic allocation of SDO channels is supported.

### **Slave assignment**

Additional object entry of CiA 302: [1F81]. Management, boot-up and troubleshooting of the individual slave modules managed by the master are configured with this list.

### **Transmission type**

The mode of a →PDO is specified via the transmission type in the communication profile of a device. CANopen provides the following transmission types for PDOs:

Synchronous: Depending on a SYNC object, transmission is either

    Acyclic: once, if process data have changed

or

    cyclic: with each reception or after a number of SYNC objects specifiable by their transmission rate.

Asynchronous: Transmission is triggered by a vendor-specific event or by an event defined via the device profile.

Remote: Transmission occurs only after being requested by another node (PDO consumer).

### **Transmission Rate**

For the cyclic-synchronous mode of a → PDO, the value of the transmission rate represents the number of synchronization messages that must have been received until the PDO is transmitted again.

### **TPDO: Transmit PDO**

→ PDO

### **VCI**

Virtual CAN Interface, driver software for IXXAT CAN boards.

### 1.4 Typographical Conventions

The following typographical conventions apply to this handbook.

Type	Meaning
V20_CN32.EDS	User input or operating system-specific elements such as file names
<b>Bitrate</b>	Lettering of an operating element or screen output
<b>TPDO</b> <b>NMTStartUp</b>	CANopen-specific term or Object name according to specification

### 1.5 Support

For additional information on IXXAT products, FAQ lists and installation tips, please refer to the support section of the IXXAT website ([www.ixxat.com](http://www.ixxat.com)), which also contains information on current product versions and available updates.

If you have any further questions after studying the information on our website and the manuals, please contact our support department. The support section on our website contains the relevant forms for your support request. In order to facilitate our support work and enable a fast response, please provide precise information on the individual points and describe your question or problem in detail.

If you would prefer to contact our support department by phone, please also send a support request via our website first, so that our support department has the relevant information available.

### 1.6 Return of defect Hardware

If it is necessary to return hardware, please download the relevant RMA form from our website and follow the instructions on this form.

In the case of repairs, please also describe the problem or fault in detail on the RMA form. This will enable us to carry out the repair quickly.



## 2 Getting Started

### 2.1 System Requirements

	Minimum requirement	Recommended
Operating system	Windows 2000	Windows 2000/XP
Processor	x86 processor 400 MHz	x86 processor 1 GHz
Main memory	64 MB RAM 100 MB temporary memory	128 MB RAM

The system requirements are mainly determined by the client application as the core functionality of the CANopen Manager API runs directly on the CAN board independently of the host computer.

### 2.2 Supported CAN Boards

At the time of writing of this manual the CANopen Manager API works together with the following active CAN board. This board features a local micro controller that runs the CANopen Manager firmware included with this product.

PC interface	CAN board	Micro controller
PCI	iPC-I XC16/PCI	Infineon XC161CJ
PCI-Express	iPC-I XC16/PCle	Infineon XC161CJ

For an up-to-date list of currently supported CAN boards, please consult the IXXAT website at [www.ixxat.com](http://www.ixxat.com).

### 2.3 VCI

The CANopen Manager API installation is based on the VCI driver software. For an up-to-date overview of the VCI version(s) required by the CANopen Manager API, please visit the IXXAT website. A free download of the VCI driver package is available in the download section of the website.



**The drivers of the CAN boards (VCI) must be installed *before* the CANopen Manager API, otherwise you will get an error when calling the API functions.**

### 2.4 Installation

Installation is carried out as follows:

- (1) Install VCI (see chapter 2.3 VCI)
- (2) Install the CAN board as described in the installation instructions of the CAN board.
- (3) Start the file `setup20.exe` which is found on the product CD and follow the instructions of the program. Generally you require administrator rights to be able to install the software successfully.

### 2.5 Flash Firmware

The CANopen Manager firmware can be executed on the CAN board from flash or from RAM. For optimal performance execution from flash is recommended which requires a specific firmware version on the CAN board. There are two different flashable firmware files provided for VCI2 and VCI3. Please see the following sections for correct flashing procedure.

#### 2.5.1 VCI2

When using VCI2, you have direct hold on the firmware type to utilise. This is due to the fact that the general download of firmware to the IXXAT CAN board RAM is configurable via Windows Control Panel. In order to use the Manager firmware from flash the download of firmware to the RAM must be *deactivated*. To do that follow these steps:

- (1) Open the system's **Control Panel**
- (2) Open **IXXAT Interfaces** applet
- (3) Choose your CAN board and expand it's attributes
- (4) Remove the checkmark at the entry **DOWNLOAD**

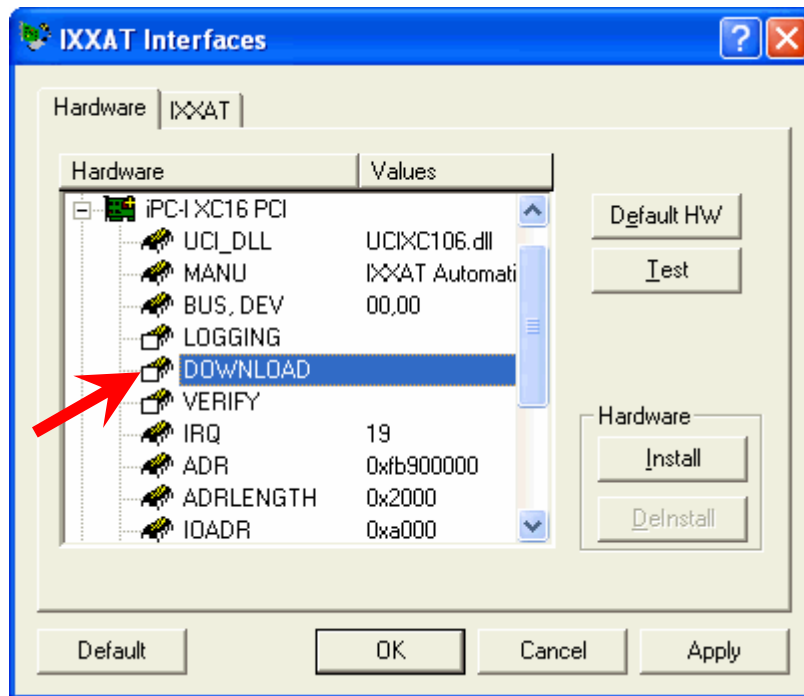
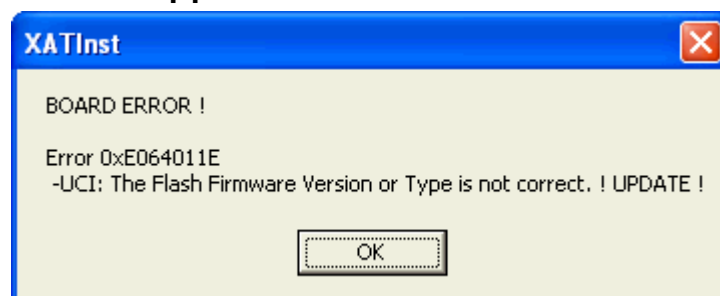


Figure 2-1: IXXAT Interfaces - Download

Usage of the Manager firmware from flash requires an appropriate board setup, otherwise yet alone the initialization function call `CMM_InitBoard` would fail. In case you did not already purchase a CAN board with flashed CANopen Manager firmware, you can flash it for yourself using the `XcFlash.exe` utility that is provided with the CANopen Manager API.



**Once a CAN board is flashed with CANopen Manager firmware it is no more useable for further VCI2 based applications. It will be displayed as outdated in the board test of the control panel's IXXAT Interfaces applet:**



**Return to VCI compatibility is accomplished by flashing of the VCI/UCI firmware provided likewise with the CANopen Manager API.**

## Getting Started

---

Start the IXXAT flash utility via the start menu entry named **Flash utility VCI2**. It is located in the installation folder sub path `\Tools`. The application window opens up (Figure 2-2).

Choose the iPC-I XC16/PCI board that is to be flashed under **Devices**. Click **Open** to select the firmware file. There are two different VCI2 compliant firmware files provided:

**ucii161f.H86** is the VCI/UCI firmware, assuring VCI2 compatibility. As it is flashed the CANopen Manager Firmware must be executed from the board's RAM.

**XATCMMFL.H86** is the CANopen Manager firmware for VCI2.

After you have chosen your firmware file, click button **Flash** to start reprogramming.

A few seconds later the flash utility reports **Device successfully flashed!**

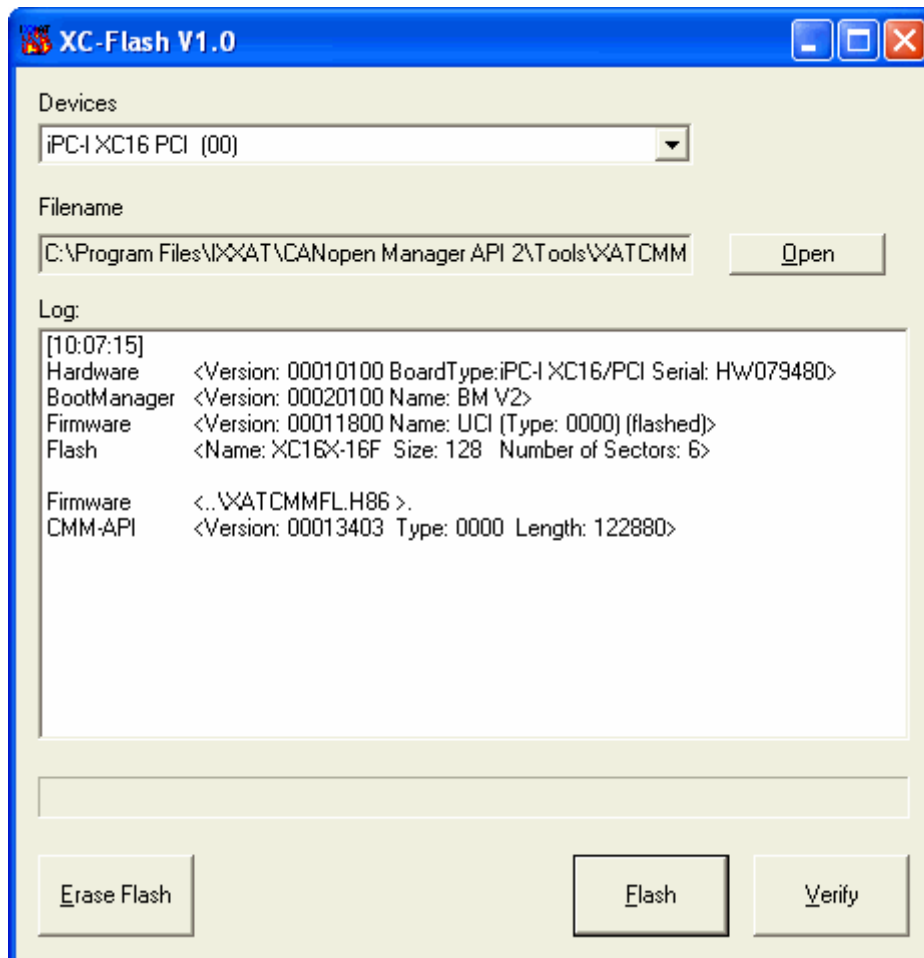


Figure 2-2: IXXAT VCI2 Windows flash utility

## 2.5.2 VCI3

VCI3 is automatically checking the currently flashed firmware on the CAN board, and loading the CANopen Manager firmware to RAM only if there is no equivalent one present there. In other words, once the VCI3 compliant Manager firmware is flashed, it will be used.

In case you did not already purchase a CAN board with flashed CANopen Manager firmware, you can flash it for yourself using the `VCI3FlashGUI.exe` utility that is provided with the VCI3.

To do so, please start it by the Windows Explorer from the VCI3 program folder. The application opens up as shown in figure 2-3.

Choose the iPC-I XC16/PCI CAN board that is to be flashed under **Device**. Click **Select Source** to select the firmware file from the installation directory `\Tools` folder:

`XATCMMFL3.H86` is the CANopen Manager firmware for VCI3.

After you have chosen your firmware file, click button **Flash** to start reprogramming.

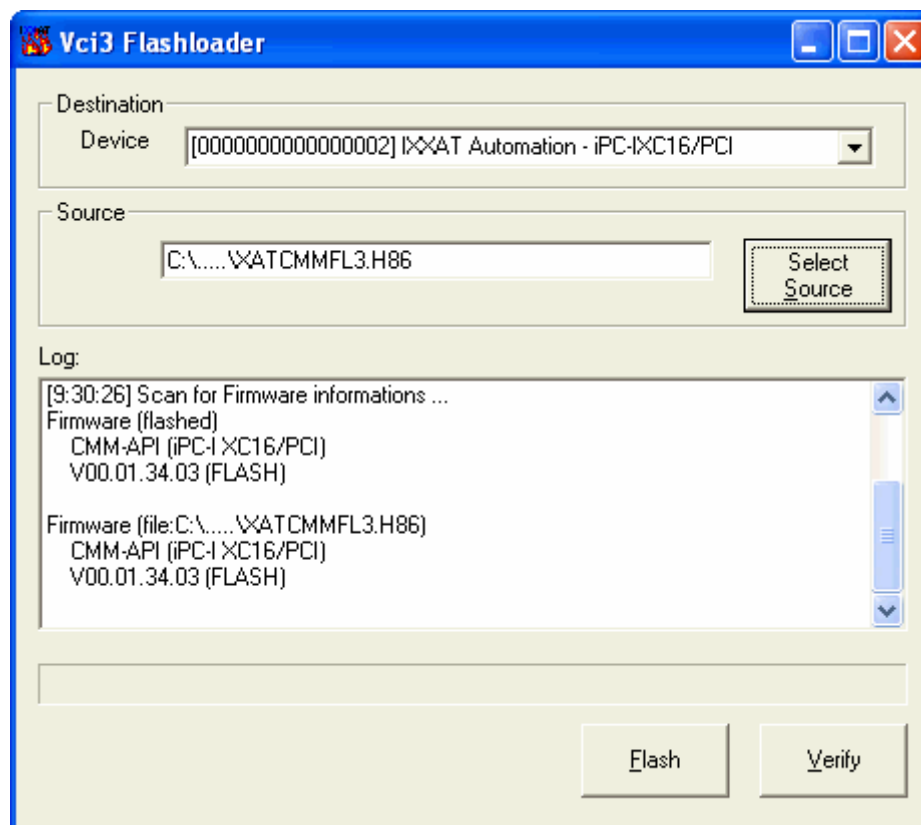


Figure 2-3: IXXAT VCI3 Windows flash utility

### 2.6 Becoming acquainted with the CANopen Manager

The product is delivered with an example application in source code for the programming language Microsoft Visual C++. This code demonstrates the use of the most common CANopen Manager API functions. The example should not be understood as a production quality, complete application.

The following functionalities are demonstrated:

- Initialization of the CANopen Manager
- Configuration of the CANopen Manager via the local object dictionary
- Communication with other CANopen nodes on the network via the CSDO interface
- Network boot-up procedure
- Execution of NMT commands
- Generation of the process image
- Data exchange via the process image
- Generation of a dynamic object dictionary entry

The typical function call order of the CANopen Manager API is outlined below. Alternatively the modes AutoConfiguration or ManualConfiguration can be used.

The first figure shows the typical function call order when the mode AutoConfiguration is used.

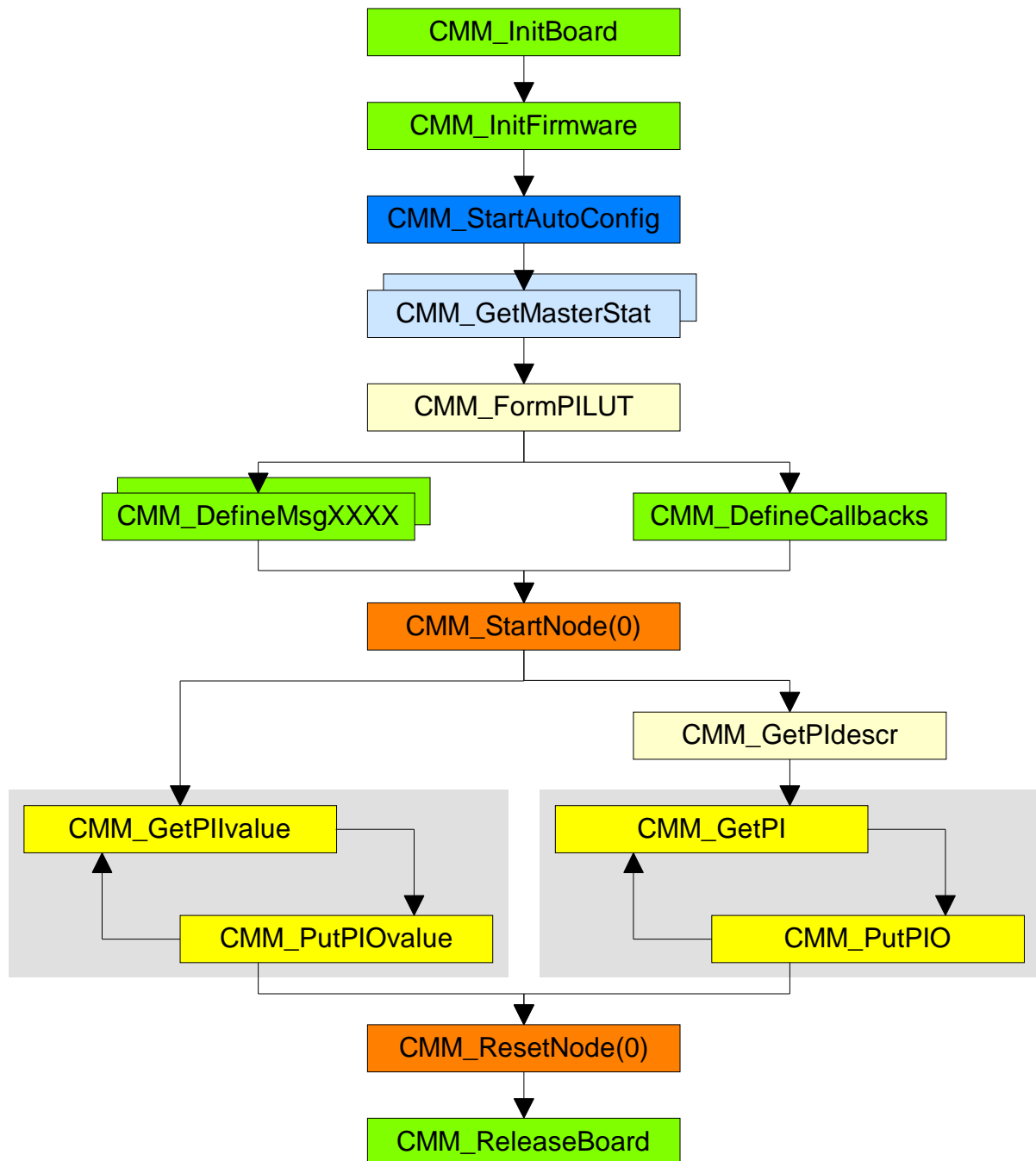


Figure 2-4: Typical function call order for mode AutoConfiguration

## Getting Started

The second figure shows the typical function call order when the mode ManualConfiguration is used.

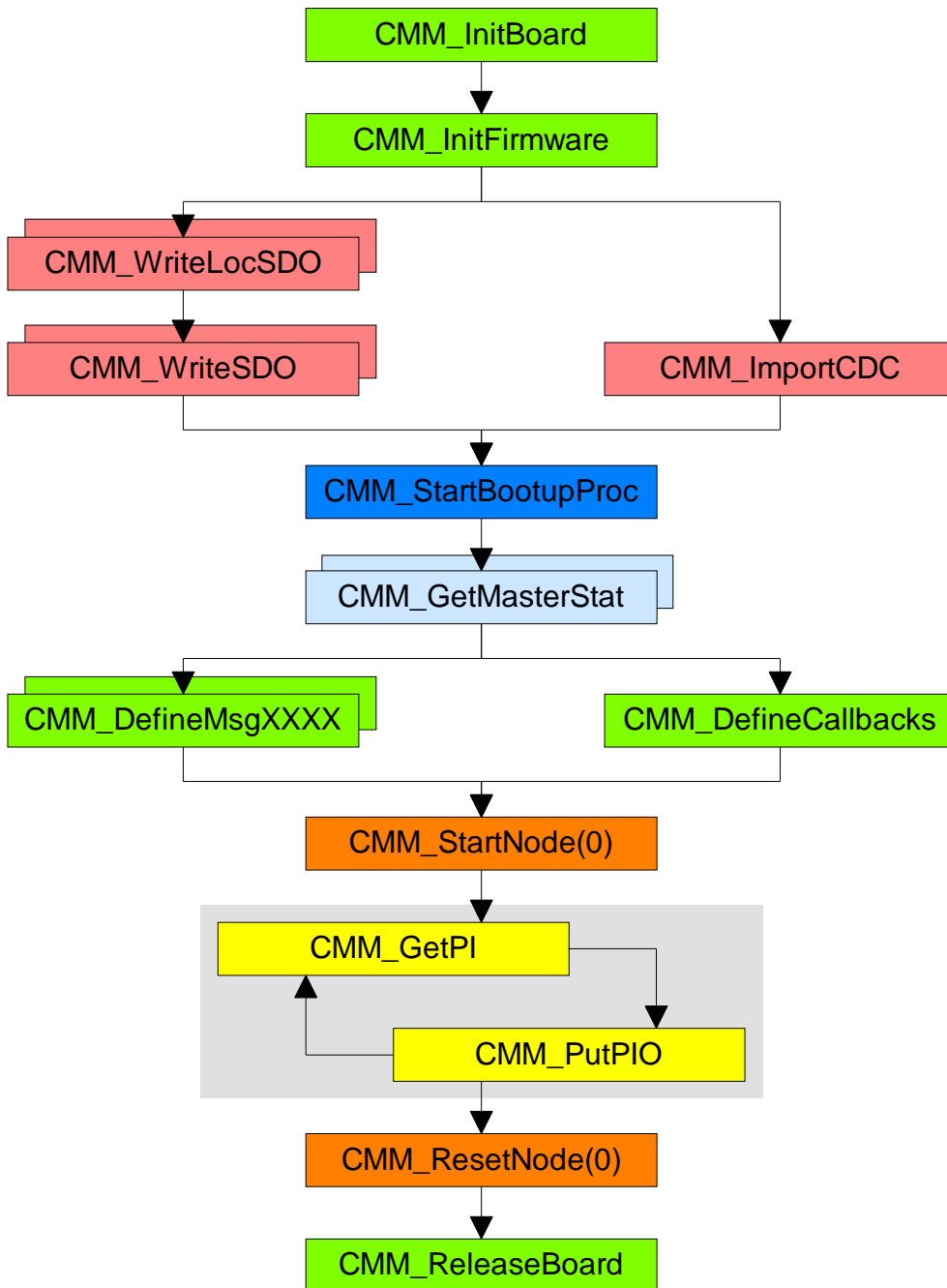


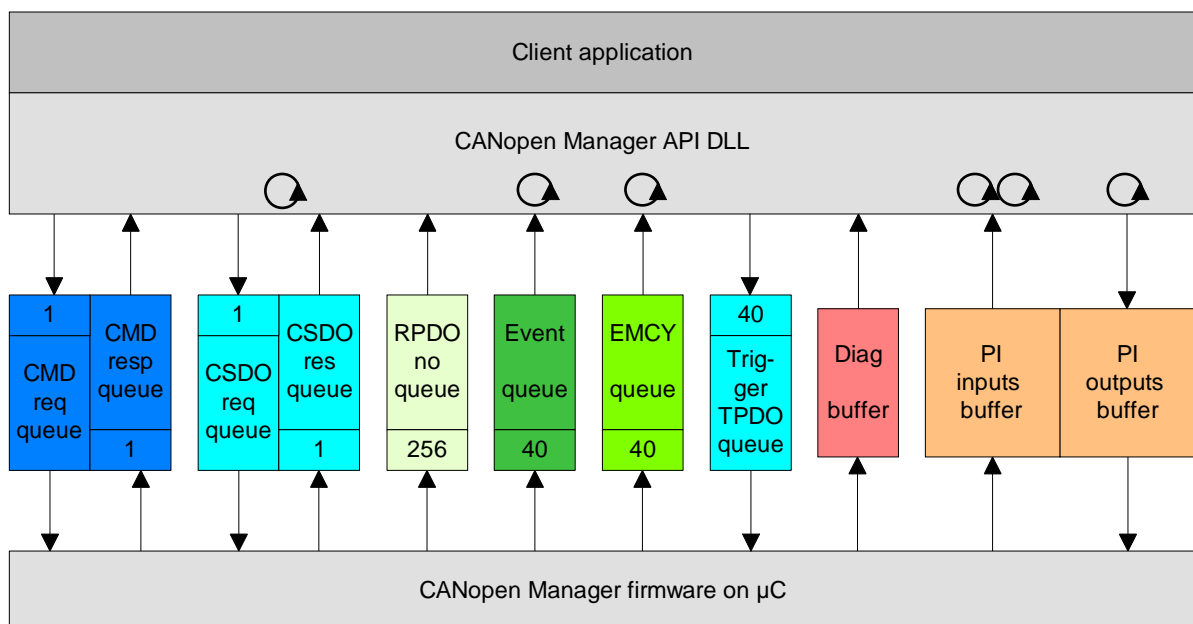
Figure 2-5: Typical function call order for mode ManualConfiguration



### 3 Overview

The local micro controller of the IXXAT CAN board executes a dedicated firmware application that implements the complete CANopen master and manager functionality. It provides the CANopen Manager API DLL with the following interfaces for the control and exchange of configuration, diagnostics and process data, see also Figure 3-1:

- Command interface to control the CANopen Manager.
- Local SDO interface to access the object dictionary of the CANopen Manager.
- CSDO interface for access to object dictionary entries of CANopen slaves on the network.
- RPDO number interface for analysis of received PDO's
- Event data interface to receive occurred events
- Emergency message interface
- Diagnostics interface for analysis of the states of CANopen Manager and CANopen slaves.
- Process image to read (PI input) and write (PI output) process variables.



**Figure 3-1: Communication between CANopen Manager API DLL and CANopen Manager firmware**

# 4 Tutorial

To allow the user to get started with the CANopen Manager API this chapter describes some typical solutions to tasks that may be addressed with this product. For reasons of simplicity, error handling is not be discussed here. The example programs presented in this chapter can be found in the sub directory **SAMPLES** within the root installation directory.

The CANopen Manager is configured completely via its object dictionary. This implies that the specification of its functional behavior is performed by means of local SDO accesses, and not by calling specific functions of the CANopen Manager API. The CANopen Manager is thus fully compatible to CiA 302 [2].

After power-on all network devices enter the NMT state Pre-operational. In this state device configuration operations are to be performed before the network can be started with the NMT command Start Remote Node and enter normal operation. During the configuration phase SDO communication objects are used primarily, both for accessing remote device as well as for accessing the local object dictionary of the CANopen Manager. Both transfer mechanisms can be realized with the CANopen Manager API.

## 4.1 Setup of the Example Network

The CANopen Manager API example assumes a network composed of four devices, three I/O modules according to the CANopen device profile CiA 401 [4], and the CANopen Manager itself. Please refer to Figure 5-1 in chapter 5.

As an initial simplification it is assumed that only the default PDOs according to the Predefined Connection Set [1] and the definitions in the device profile CiA 401 are supported. All process data produced and consumed by the CANopen slave devices are mapped into the process image of the CANopen Manager.

The I/O modules are configured to use node-IDs 10, 11, and 12, the CANopen Manager uses node-ID 127.

For network-wide device monitoring the heartbeat mechanism is used.

## 4.2 Initialization of the CANopen Manager

- (1) Selection of a CAN board with `CMM_InitBoard()`  
`CMM_InitBoard()` receives the board type (`pBoardtype`) and a system-wide unique board identifier (`pBoardID`) as arguments. On successful

execution of the board initialization a board handle (**phBoard**) is returned to the caller which identifies the CAN board in all subsequent API calls. At most 4 CAN boards may be used in parallel:

```
tCMM_HANDLE g_hBoard;
GUID        g_sBoardtype = GUID_IPCIXC16PCI_DEVICE;
GUID        g_sBoardID   = CMM_1stBOARD;

CMM_InitBoard( &g_hBoard,
               &g_sBoardtype,
               &g_sBoardID );
```

- (2) **CMM\_InitFirmware()** initializes the CANopen Manager firmware. This function sets the bit rate (**Baudrate**) of the CAN network and the node-ID of the CANopen Manager device itself (**NodeNo**). Also the handshake interval for the handshaking and the reactions of the Manager firmware on an handshake timeout are set (see section 5.17, Handshaking). The value 0 for the handshake interval means the handshaking will not be watched by the Manager firmware. It is recommended to set the **InitMode** argument to **COP\_k\_RESETPNODE** (see sections 5.10, Initialization of the CANopen Manager and 11.1.4, **CMM\_InitFirmware**) on the first call to **CMM\_InitFirmware()**:

```
BYTE bInitMode      = COP_k_RESETPNODE;
BYTE g_bBaudrate    = CMM_BAUDRATE_125;
BYTE g_bNodeNo      = 127;
WORD g_wHsInterval  = 0;
WORD g_wHsReaction  = 0;

CMM_InitFirmware( g_hBoard,
                  bInitMode,
                  g_bBaudrate,
                  g_bNodeNo,
                  g_wHsInterval,
                  g_wHsReaction );
```

The CANopen Manager is now initialized and may commence operation. The next steps cover the configuration of the CANopen Manager functionality, the declaration of CANopen slave devices connected to the network, and the initialization of the process image.

### 4.3 Configuration of the CANopen Manager via its local Object Dictionary

The CANopen Manager integrates a complete object dictionary, that is located in the volatile memory of the CAN board. An overview of the available objects can be found in chapter 5, CANopen Manager Firmware, a complete description is contained in the device description file (EDS) that is shipped with this product.

## Tutorial

---

For an explanation of the functionality linked to the individual entries please consult the respective CANopen specifications [1][2][3][4][5].

A write access to the object dictionary of the CANopen Manager is performed with the API function `CMM_WriteLocSDO()`, for a read access the function `CMM_ReadLocSDO()` is available.

- (1) Configuration of the heartbeat period of CANopen Manager [1017]  
The CANopen Manager shall transmit a heartbeat message twice a second. According to CiA 301 this requires to write the parameter value 500 (ms) as an UNSIGNED16 into the local object dictionary entry [1017]:

```
BYTE  abTxdata[2] = {0xF4, 0x01}; // 500 milliseconds
DWORD dwAbortcode = 0;

CMM_WriteLocSDO( g_hBoard,
                 0x1017, 0x00, // Producer Heartbeat Time
                 sizeof(abTxdata),
                 abTxdata,
                 &dwAbortcode );
```

- (2) Registration of all CANopen nodes available in the network with the NMT error control service of the CANopen Manager [1016]  
The use of the heartbeat mechanism as the network-wide device monitoring mechanism requires the configuration of the object **consumer heartbeat time** [1016] in the object dictionary of the CANopen Manager. This object is specified in CiA 301 [1]. In the example configuration the slave devices shall produce their heartbeat message at intervals of 400ms. The corresponding configuration steps on the slave devices, i.e. writing to object [1017], are performed by the CANopen Manager automatically after their actual detection, by using a given Device Configuration File (DCF) in binary concise format. This is discussed in detail after the following step. A producer heartbeat time of 400ms requires the corresponding consumer time to be set to about 500ms to cope for potential delays of the heartbeat message by the CAN arbitration process. The **consumer heartbeat time** object is coded as an UNSIGNED32 value, with heartbeat time in the LSW, and the corresponding node-ID of the monitored device in the MSW.

```
BYTE  abTxdata[4] = {0xF4, 0x01, 0x00, 0x00}; // 500 milliseconds
DWORD dwAbortcode = 0;

abTxdata[2] = 10; // Slave Node 10
CMM_WriteLocSDO( g_hBoard,
                 0x1016, 0x01, // 1st Consumer Heartbeat Time
                 sizeof(abTxdata),
                 abTxdata,
                 &dwAbortcode );

abTxdata[2] = 11; // Slave Node 11
```

```

CMM_WriteLocSDO( g_hBoard,
                 0x1016, 0x02,    // 2nd Consumer Heartbeat Time
                 sizeof(abTxdata),
                 abTxdata,
                 &dwAbortcode );

abTxdata[2] = 12;                // Slave Node 12
CMM_WriteLocSDO( g_hBoard,
                 0x1016, 0x03,    // 3rd Consumer Heartbeat Time
                 sizeof(abTxdata),
                 abTxdata,
                 &dwAbortcode );

```

- (3) Adjustment of the CANopen Manager's boot-up behavior [1F80]  
 In the power-on settings, CANopen Manager behaves most defensive and passive regarding the NMT Master functionality. The belonging object dictionary entry is **NMTStartup** [1F80]. In order to allow the Manager to freely start its assigned slaves, Bit 3 of the object needs to be cleared:

```

BYTE  abValue[4] = {0};
DWORD len        = sizeof(abValue);
DWORD dwAbortcode = 0;

if( CMMERR_OK == CMM_ReadLocSDO( g_hBoard,
                                0x1F80, 0x00,    // NMT Startup
                                &len, abValue,
                                &dwAbortcode ) )
{
    abValue[0] ^= 0x08;                // Clear Bit 3
    CMM_WriteLocSDO( g_hBoard,
                    0x1F80, 0x00,    // NMT Startup
                    len, abValue,
                    &dwAbortcode );
}

```

- (4) Registration of all slaves devices with the CANopen Manager [1F81]  
 According to the CiA 302 [2] specification, the way slave devices are managed by the CANopen Manager is configured in object **SlaveAssignment** [1F81]. In the most simple case a CANopen slave is declared as optional device and is booted automatically by the CANopen Manager, corresponding to a bit pattern **0x05**. It is recommended to register not only currently installed devices, but also those that might be added to the network at a later point in time. Note that those devices have to be declared as optional devices. The object [1F81] is specified as an array of UNSIGNED32, of which the upper three byte are not interpreted if the network is monitored by means of heartbeat:

```

BYTE  i;
BYTE  abTxdata[4] = {0x05, 0x00, 0x00, 0x00}; // Boot optional slave
DWORD dwAbortcode = 0;

```

```
for( i = 1 ; i <= 127 ; i++ )
    CMM_writeLocSDO( g_hBoard,
                    0x1F81, i, // Slave Assignment Node i
                    sizeof(abTxdata),
                    abTxdata,
                    &dwAbortcode );
```

- (5) Providing the device configuration files of all slaves devices [1F22]
- Any slave declared in the **SlaveAssignment** list [1F81] will be included in the regular network boot-up process according to the CiA 302 [2] specification. At the right time of the boot-up process, the CANopen Manager will write an individual device configuration to each node deposited in object **Concise DCF** [1F22]. This device configuration is given by the application and usually generated by a network configuration tool in form of a binary device configuration file. As it contains only the essential objects differing from their default (shipping) value, it is referred to as concise DCF. In the most simple case it consists of only a few entries setting up the basic project settings like sync cycle and node monitoring interval. Prior to downloading the concise DCF contents to the slaves, to obtain a clearly defined starting point before generating the process image in the CANopen Manager, all slave devices are reset to their factory settings. All this is performed by the CANopen Manager during the boot-up procedure. Since there have been devices registered to the error control service of the CANopen Manager in one of the previous steps, there must be a minimal concise DCF written to [1F22] configuring the producer heartbeat interval of the slave. The corresponding cDCF is 13 bytes long and addresses the slave object [1017sub0] with the value 400 (0x190), so it is ensured the Manager will receive a heartbeat of each configured slave within the configured consumer heartbeat time of 500 milliseconds:

```
BYTE i;
// Producer Heartbeat time 400ms
BYTE abTxdata[13] = {0x01, 0x00, 0x00, 0x00, // 1 entry
                    0x17, 0x10, 0x00, // Prod HB
                    0x02, 0x00, 0x00, 0x00, // WORD value
                    0x90, 0x01}; // 400ms
DWORD dwAbortcode = 0;

for( i = 1 ; i <= 127 ; i++ )
    CMM_writeLocSDO( g_hBoard,
                    0x1F22, i, // Concise DCF Node i
                    sizeof(abTxdata),
                    abTxdata,
                    &dwAbortcode );
```

## 4.4 Generation of Process Images

Next the configuration of the first four RPDOs and TPDOs of all slave devices is read and exactly matching network variables are generated in the process image of the CANopen Manager. The relation between the PDOs on the CANopen Manager and the corresponding PDOs on the CANopen Slaves is depicted in Figure 4-1 considering as example a device's first RPDO and TPDO.

For the readout of the individual nodes' PDO configuration, the CSDO (**Client Service Data Object**) interface of the CANopen Manager with the functions `CMM_ReadSDO()` and `CMM_WriteSDO()` is utilized. This CANopen service provides a unidirectional individual communication channel from the Manager to a slave node.

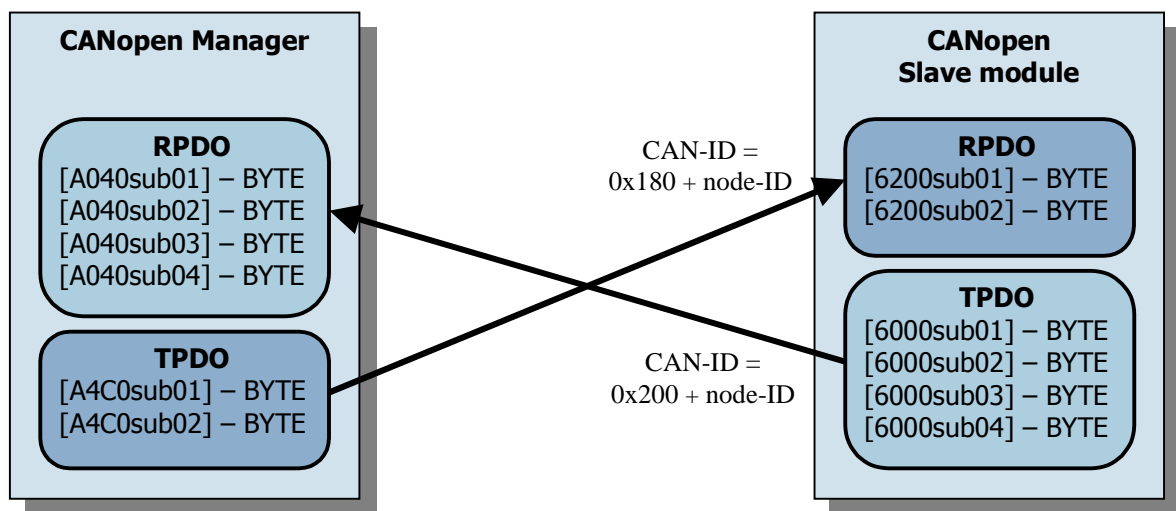


Figure 4-1: Schematic linking of process data objects of the CANopen Manager and a CANopen slave

- (1) Configuration of the RPDOs on the CANopen Manager  
 In a first step the PDO mapped objects of the first four **transmit PDOs** of the CANopen slave device are evaluated and for each object mapped in one of the TPDOs a corresponding **network variable** is created in the **process image inputs**. The indices under which the network variables can be addressed within the object dictionary are specified in CiA 405 [5]. From the index of a network variable in the object dictionary follows implicitly the offset of the received process value within the process image. Note that for the allocation of entries in the process image the principle of overlaid network variable applies. Please see section 6.1.3, Overlaid Network Variables for further details.

To determine the object dictionary index of a process variable based on its data type and its offset in the process image a ancillary function `CMM_NVaddrFromPIoffset()` is used for which the prototype is declared in the header `XatCMMutil.h`.

```
void EstablishRPDOLinks( BYTE aNodeNo,
                       WORD* aRPDO, WORD *aPIIoffset )
{
    tCMM_ERROR res;
    DWORD      dwRxlen;
    BYTE       abRxdata[4];
    WORD       wPDO = 0;
    WORD       wPDOvalid = 0;
    DWORD      dwCANID;
    BYTE       bMappingCnt;
    WORD       wNVmain;
    BYTE       bNVsub;
    DWORD      dwRPDOMapping;

    while( wPDO < 4 )           // Process only the first 4 PDOS
    {
        // Read the TPDO's CAN-ID from slave
        dwCANID = 0;
        dwRxlen = sizeof(dwCANID);
        res = CMM_ReadSDO( g_hBoard, aNodeNo,
                          CMM_DEFAULT_SDO, SDOMODE_SEGMENTED,
                          0x1800+wPDO, 0x01, //PDO_CommPar.CAN-ID
                          &dwRxlen, (BYTE*)&dwCANID, NULL );

        if( (CMMERR_OK == res) // PDO exists
            && (0 == (dwCANID & 0x80000000)) ) // PDO is valid
        {
            // Read TPDO number of mapped objects
            bMappingCnt = 0;
            dwRxlen = sizeof(bMappingCnt);
            CMM_ReadSDO( g_hBoard, aNodeNo,
                          CMM_DEFAULT_SDO, SDOMODE_SEGMENTED,
                          0x1A00+wPDO, 0x00, // PDO_Mapping.count
                          &dwRxlen, &bMappingCnt, NULL );

            if( (bMappingCnt <= 64) && (bMappingCnt > 0) )
            {
                // Process all TPDO mapped objects
                for( BYTE i = 1 ; i <= bMappingCnt ; i++ )
                {
                    // Read TPDO mapped object
                    dwRxlen = sizeof(abRxdata);
                    CMM_ReadSDO( g_hBoard, aNodeNo,
                                CMM_DEFAULT_SDO, SDOMODE_SEGMENTED,
                                0x1A00+wPDO, i, //PDO_Mapping.i-th_obj
                                &dwRxlen, abRxdata, NULL );

                    // Calculate a NV matching the mapped object
                    CMM_VNaddrFromPIoffset( wNVmain, bNVsub,
                                             *aPIIoffset,
                                             abRxdata[0]/8, // size(bytes)
                                             true ); // PII
                    dwRPDOMapping = (wNVmain << 16) | (bNVsub << 8)
                                     | abRxdata[0];
                    *aPIIoffset += abRxdata[0]/8;

                    // write NV as mapped object to local RPDO
                    CMM_WriteLocSDO( g_hBoard,
                                     0x1600+wPDO+*aRPDO, i,
                                     sizeof(dwRPDOMapping),
```



```

        (BYTE*)&dwRPDOMapping, NULL );
    }

    // write number of mapped objects to local RPDO
    CMM_WriteLocSDO( g_hBoard,
                    0x1600+WPDO+*aRPDO, 0x00,
                    sizeof(bMappingCnt),
                    &bMappingCnt, NULL );

    // write the TPDO's CAN-ID to local RPDO
    dwCANID &= 0x1FFFFFF; // Mask in CAN-ID
    CMM_WriteLocSDO( g_hBoard,
                    0x1400+WPDO+*aRPDO, 0x01,
                    sizeof(dwCANID),
                    (BYTE*)&dwCANID, NULL );

    wPDOvalid++;
}
}
}
WPDO++; // Next PDO
}
*aRPDO += wPDOvalid;
}

```

## (2) Configuration of TPDOs of the CANopen Manager

The TPDOs of the CANopen Manager are configured equivalent to the procedure with the RPDOs, as outlined in the paragraph above. The RPDOs of the slave devices are evaluated, and for each object mapped into one of the RPDOs of the slave a corresponding **network variable** is created in the **process image outputs**.

Except for the object dictionary addresses of the PDOs and the last parameter of the ancillary function `CMM_VNaddrFromPIOffset()` the function `EstablishTPDOLinks` listed below is identical to `EstablishRPDOLinks`, see (1).

```

void EstablishTPDOLinks( BYTE aNodeNo,
                        WORD* aTPDO, WORD *aPIOffset )
{
    tCMM_ERROR res;
    DWORD      dwRxlen;
    BYTE       abRxdata[4];
    WORD       wPDO = 0;
    WORD       wPDOvalid = 0;
    DWORD      dwCANID;
    BYTE       bMappingCnt;
    WORD       wNVmain;
    BYTE       bNVsub;
    DWORD      dwTPDOMapping;

    while( wPDO < 4 ) // Process only the first 4 PDOs
    {
        // Read the RPDO's CAN-ID from slave
        dwCANID = 0;
        dwRxlen = sizeof(dwCANID);
        res = CMM_ReadSDO( g_hBoard, aNodeNo,
                          CMM_DEFAULT_SDO, SDOMODE_SEGMENTED,
                          0x1400+wPDO, 0x01, //PDO_CommPar.CAN-ID
                          &dwRxlen, (BYTE*)&dwCANID, NULL );
    }
}

```

```
if( (CMMERR_OK == res) // PDO exists
&& (0 == (dwCANID & 0x80000000)) ) // PDO is valid
{
    // Read RPDO number of mapped objects
    bMappingCnt = 0;
    dwRxlen = sizeof(bMappingCnt);
    CMM_ReadSDO( g_hBoard, aNodeNo,
                 CMM_DEFAULT_SDO, SDOMODE_SEGMENTED,
                 0x1600+wPDO, 0x00, // PDO_Mapping.count
                 &dwRxlen, &bMappingCnt, NULL );

    if( (bMappingCnt <= 64) && (bMappingCnt > 0) )
    {
        // Process all RPDO mapped objects
        for( BYTE i = 1 ; i <= bMappingCnt ; i++ )
        {
            // Read RPDO mapped object
            dwRxlen = sizeof(abRxdata);
            CMM_ReadSDO( g_hBoard, aNodeNo,
                        CMM_DEFAULT_SDO, SDOMODE_SEGMENTED,
                        0x1600+wPDO, i, //PDO_Mapping.i-th_obj
                        &dwRxlen, abRxdata, NULL );

            // Calculate a NV matching the mapped object
            CMM_VNaddrFromPIoffset( wNVmain, bNVsub,
                                   *aPIOffset,
                                   abRxdata[0]/8, //size (bytes)
                                   false ); // PIO
            dwTPDOMapping = (wNVmain << 16) | (bNVsub << 8)
                            | abRxdata[0];
            *aPIOffset += abRxdata[0]/8;

            // write NV as mapped object to local TPDO
            CMM_WriteLocSDO( g_hBoard,
                            0x1A00+wPDO+*aTPDO, i,
                            sizeof(dwTPDOMapping),
                            (BYTE*)&dwTPDOMapping, NULL );
        }

        // write number of mapped objects to local TPDO
        CMM_WriteLocSDO( g_hBoard,
                        0x1A00+wPDO+*aTPDO, 0x00,
                        sizeof(bMappingCnt),
                        &bMappingCnt, NULL );

        // write the RPDO's CAN-ID to local TPDO
        dwCANID &= 0x1FFFFFF; // Mask in CAN-ID
        CMM_WriteLocSDO( g_hBoard,
                        0x1800+wPDO+*aTPDO, 0x01,
                        sizeof(dwCANID),
                        (BYTE*)&dwCANID, NULL );

        wPDOvalid++;
    }
}

wPDO++; // Next PDO
*aTPDO += wPDOvalid;
}
```

## 4.5 CANopen Network Boot-up

At this stage the configuration of the CANopen Manager and the slave devices is completed. All network devices are registered with the CANopen Manager, the

network-wide device monitoring is set to a heartbeat interval of 400ms, and the relations between the process input and output variables of all devices have been established.

Next the network is started (see also section 5.3, Boot-up Procedure). The boot-up procedure passes through multiple internal states of the CANopen Manager which are indicated to application by means of the CANopen Manager state buffer. See section 7.1.1, State of the CANopen Manager, for details. After having completed the boot-up procedure the CANopen Manager has full control of the network and exchanges PDOs with the slave devices. The CANopen Manager also keeps track of the NMT state of the slave devices (see section 7.2, Slave Diagnostics), and handles the device monitoring (see section 7.3, Emergency Statistic and History).

- (1) Start of the boot-up procedure  
Triggered by a call to the API function `CMM_StartBootupProc()`, the CANopen Manager starts the network and transfers it into the NMT state **operational**:

```
CMM_StartBootupProc( g_hBoard );
```

- (2) Monitoring the boot-up procedure  
It is recommended for the application to cyclically monitor the state of the CANopen Manager until a final state is detected in the low byte of `wMasterManagerState`. Stable final states are `CLEAR`, `RUN`, and `FATAL_ERROR`, however only if the state `RUN` is attained the network has been started successfully. In the final state `CLEAR` the boot-up procedure has been completed successfully as well, however the CANopen Manager was not configured to start the network automatically and is waiting for the command `CMM_StartNode()` by the application to explicitly start the network. In the case the CANopen Manager has assumed the state `FATAL_ERROR` a serious error has been detected. This error case has to be analyzed by the user with the help of the information contained in the fields `wMasterManagerState` and `wGlobalEvents` before any further processing is permitted. In this case starting the network is not possible.

```
tCMM_ERROR res;  
WORD      wMasterManagerState  
WORD      wGlobalEvents;  
WORD      wConfigBits;  
  
do  
{  
    sleep(222);  
    res = CMM_GetMasterStat( g_hBoard,  
                             &wMasterManagerState,  
                             &wGlobalEvents,  
                             &wConfigBits );
```

```
}
while((CMMERR_OK == res)
&& (CLEAR != (wMasterManagerState & 0xF0))
&& (RUN != (wMasterManagerState & 0xF0))
&& (FATAL_ERROR != (wMasterManagerState & 0xF0)));
```

(3) Explicit start of the network

If the CANopen Manager has assumed the state `CLEAR`, the network has to be started explicitly by the application. Before the network is started the application may verify the state of the individual CANopen slave devices:

```
if( CLEAR == (wMasterManagerState & 0xF0) )
{
    tCMM_SLAVEFLAGS fAssigned = {0};
    tCMM_SLAVEFLAGS fConfigured = {0};
    tCMM_SLAVEFLAGS fMismatch = {0};
    tCMM_SLAVEFLAGS fEmergency = {0};
    tCMM_SLAVEFLAGS fOperational = {0};
    tCMM_SLAVEFLAGS fStopped = {0};
    tCMM_SLAVEFLAGS fPreOperational = {0};

    res = CMM_GetSlavesStat( g_hBoard,
                            &fAssigned,
                            &fConfigured,
                            &fMismatch,
                            &fEmergency,
                            &fOperational,
                            &fStopped,
                            &fPreOperational );

    if( CMMERR_OK == res )
        CMM_StartNode( g_hBoard, 127 ); // NMT: Start entire network
}
```

## 4.6 Data Exchange via the Process Image

As soon as the network has been started the main task of the application has to cyclically inquire the entries in the process image, to process the received data, and to update the process image outputs.

Further the application processes the monitoring of the network status and will react on possible error situations that cannot be handled by the CANopen Manager itself. Alternatively to the polling of the network state, the application may register callback functions with the CANopen Manager, or it may be notified via user defined Windows messages.

Generally these events can be classified into five different categories: Data change in the process image input, change of state of the CANopen Manager, change of NMT state of a slave device, reception of an emergency message from a slave device, and finally unexpected events.

## (1) Declaration of change of state messages

In the example discussed here it is preferred to declare callback functions to notify the application of state changes of the slave devices and other events. For changes within the process image no callback functions are required as they are cyclically polled, as discussed in the following paragraph. Please note that the callback functions execute in a thread independent of the thread of the main application, as they are called from within the CANopen Manager API DLL. In this context it is mentionable that the polling threads - required by the callback mechanism - cannot proceed their processing as long as the application executes the corresponding callback functions.

```

void CALLBACK CB_CMMslavesChange( tCMM_HANDLE hBoard,
                                  DWORD         dwSource,
                                  DWORD         dwRes )
{
    // Static variables, they keep the reference values
    static tCMM_SLAVEFLAGS  asStatSlavesState[7] = {0};

    // Current values
    tCMM_SLAVEFLAGS         asCurrSlavesState[7] = {0};

    CMM_GetSlavesStat( hBoard,
                      &asCurrSlavesState[0], //Assigned
                      &asCurrSlavesState[1], //Configured
                      &asCurrSlavesState[2], //Mismatch
                      &asCurrSlavesState[3], //Emergency
                      &asCurrSlavesState[4], //Operational
                      &asCurrSlavesState[5], //Stopped
                      &asCurrSlavesState[6], //PreOperational );

    //..
    // Detect changes
    //..

    // Adapt new values
    CopyMemory( asStatSlavesState,
                asCurrSlavesState,
                sizeof(asStatSlavesState) );
}

void CALLBACK CB_CMMnotification( tCMM_HANDLE hBoard,
                                  DWORD         dwSource,
                                  DWORD         dwRes )
{
    tCMM_ERROR res = CMMERR_OK;
    BYTE       bEvtType, bEvtData1, bEvtData2;

    // Read out all the events
    do
    {
        bEvtType = bEvtData1 = bEvtData2 = 0;
        res = CMM_GetEvent( hBoard, &bEvtType,
                           &bEvtData1, &bEvtData2,
                           NULL, NULL );
        if( CMMERR_NO_OBJECT != res )
        {
            //..
            // Interpret and process the Event
            //..
        }
    }
}

```

```
    }
    while( CMMERR_OK == res )
}

void CALLBACK CB_CMMEmergencyMsg( tCMM_HANDLE hBoard,
                                  DWORD         dwSource,
                                  DWORD         dwRes )
{
    tCMM_ERROR res = CMMERR_OK;
    BYTE       bNodeNo, bErrRegister, abErrField[5];
    WORD       wErrCode;

    // Read out all the Emergency messages
    do
    {
        bNodeNo = bErrRegister = 0;
        wErrCode = 0;
        ZeroMemory( abErrField, sizeof(abErrField) );
        res = CMM_GetEmergencyObj( hBoard, &bNodeNo,
                                   &wErrCode, &bErrRegister,
                                   abErrField );

        if( CMMERR_NO_OBJECT != res )
        {
            //..
            // Interpret and process the EMCY msg
            //..
        }
    }
    while( CMMERR_OK == res )
}

// Define Callbacks
CMM_DefineCallbacks(
    g_hBoard,
    NULL,          // Change of Process Image Inputs
    NULL,          // Change of MasterManager status
    CB_CMMSlaveChange, // Change of Slaves state
    CB_CMMnotification, // Event notification
    CB_CMMEmergencyMsg ); // Received emergency messages
```

### (2) Processing of the process image inputs

The application calls the function `CMM_GetPIentry()`, to retrieve a subset of the process image from the CANopen Manager. As argument the application passes the process image type, an offset inside the process image, the length of the requested subset and a pointer to a buffer to where the current data content shall be copied to by the API function.

```
DWORD dwLengthPII;
BYTE  abPII[128] = {0};

while( !Terminated )
{
    dwLengthPII = sizeof(abPII);
    CMM_GetPIentry( g_hBoard,
                   PITYPE_INPUTS,
                   0, &dwLengthPII, // First 128 Bytes of PII
                   abPII, NULL );

    //..
    // Process the Processimage Inputs
    //..
}
```

```

    //..
    // write the Processimage Outputs
    //..
}

```

(3) Updating the process image outputs

At the end of a PLC cycle data entries in the process image output are updated by means of the API function `CMM_PutPIOentry()`. Calling this function causes the application data buffer - pointed to by the call parameter `*pPIOentry` - to be copied into the **process image outputs** of the CANopen Manager. The corresponding section of the process image is marked as updated, so that the firmware can perform a byte-by-byte compare and transmit the corresponding TPDOs.

```

DWORD dwLengthPIO;
BYTE  abPIO[128] = {0};

while( !Terminated )
{
    //..
    // Read the Processimage Inputs
    //..
    //..
    // Process the Processimage Inputs
    //..
    dwLengthPIO = sizeof(abPIO);
    CMM_PutPIOentry( g_hBoard,
                    0, dwLengthPIO, // First 128 Bytes of PIO
                    abPIO );
}

```

This concludes the commissioning of a CANopen network with the CANopen Manager API and the subsequent processing of input/output variables in the context of this tutorial. In the following sections specific aspects are introduced.

## 4.7 Auto Configuration Mode

If the alternative function `CMM_StartAutoConfig()` is used instead of `CMM_StartBootstrapProc()`, the detection and the configuration of the network as described in sections 4.3 to 4.4, is carried out automatically by the CANopen Manager firmware.

In this case the following issues should be considered:

- Network variables are created in the process images for all input/output parameters identified in the slave devices. This may result in an undesired increase of the process image size, as all configured PDOs, possibly more than the first four default PDOs as described in the tutorial sections, are evaluated.

- During the automatic creation of the network variables, it is not considered that the same remote object could be mapped multiple times. This results in a separate network variable for each mapped object.
- Due to limitations of the CANopen SDO protocol, the CANopen Manager firmware cannot determine the exact data types of the process data mapped into the PDOs of the slave devices, but only their size in Bytes. The firmware therefore creates network variables based on corresponding data size using unsigned integral data types.
- Following the automatic generation of the process images, i.e. immediately after the call to `CMM_StartAutoConfig()` the application should inquire a description of the identified slave input/output data. For this purpose the CANopen Manager API provides the functions `CMM_FormPILOT()` and `CMM_GetPIdescr()`.

A detailed discussion of the auto configuration process is given in sections 5.9, Auto Configuration Mode and 8.1.4, Auto Configuration.

### 4.8 Dynamic Generation of Object Dictionary Entries

The CANopen Manager can create application specific object dictionary entries. These entries, which may be of a size up to 8 bytes, are managed within the process images. Access to such an entry is possible by means of local SDO accesses, whereupon the application acts as SDO server, or by PI access. If it is required the entry can also be mapped into a PDO.

When creating such a dynamic object dictionary entry with the function `CMM_CreateODentry()`, the application has to specify index and sub-index, data and access type, as well as the default value of the object. In addition the application has to indicate at which offset within the process image this dynamic object shall be located. The decision whether the object is located in the process image inputs or outputs is based on the access type specified for the object. Dynamic objects with access type `ACCESSTYPE_RWW` are located in the process image inputs, those with `ACCESSTYPE_RO` will be located in the process image outputs. It is in the responsibility of the application to avoid address conflicts in the process images between network variables and dynamically generated application objects.

Dynamic objects may only be created in the CANopen Manager state `RESET`, i.e. before the network is started.

- (1) Dynamic generation of an application specific object dictionary entry



```

BYTE  abInitialValue[8] = {0x11, 0x22, 0x33, 0x44,
                           0x55, 0x66, 0x77, 0x88};

CMM_CreateODentry( g_hBoard,
                  0x5E00, 0x00, // Mainindex, SubIndex
                  DATATYPE_UNSIGNED64,
                  ACESSTYPE_RWW,
                  PDOMAPPING_SUPPORTED,
                  abInitialValue,
                  0x680 ); // offset in the PI

```

(2) Writing an application specific object dictionary entry

A dynamically created object dictionary entry can be accessed in two different ways, either via its entry in the local object dictionary, or via the process image. If accessing the entry in the process image inputs or outputs, the offset of the entry in the corresponding process image has to be known, however this kind of access is optimized in terms of performance. The access via the local object dictionary is easier to manage as only index and sub-index of the object have to be known. Below an example of a write access by means of local SDO is listed:

```

BYTE  abTxdata[8] = {0x08, 0x07, 0x06, 0x05,
                    0x04, 0x03, 0x02, 0x01};

CMM_WriteLocSDO( g_hBoard,
                 0x5E00, 0x00, // Mainindex, SubIndex
                 sizeof(abTxdata), abTxdata, NULL );

```

(3) Reading an application specific object dictionary entry

Contrary to the write access discussed above, now the process image inputs area, in which the object is located, is read directly:

```

DWORD dwLength;
BYTE  abDataPII[8] = {0};

dwLength = sizeof(abDataPII);
CMM_GetPIentry( g_hBoard,
                PITYPE_INPUTS,
                0x680, // offset in the PI
                &dwLength, abDataPII, NULL );

```

# 5 CANopen Manager Firmware

## 5.1 Overview

The CANopen Manager simplifies monitoring and control of CANopen networks. It enables client applications to configure slave modules and to react individually to failures of modules. With the CANopen Manager it is possible to manage CANopen networks with up to 127 network devices. Figure 5-1 shows the typical structure of a CANopen network, in which the CANopen Manager is used as the central control device. The slave modules themselves do not have any functions to control the network. However, with the CANopen Manager they are able to control the network via the **RequestNMT** object [1F82].

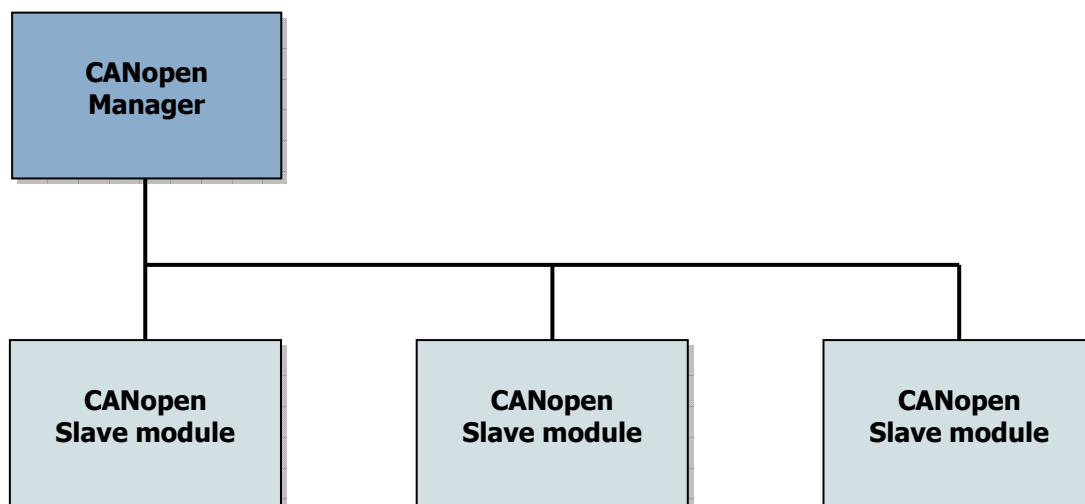


Figure 5-1: Typical structure of a CANopen network

The software concept of the CANopen Manager is based on a separation of CANopen firmware and client application via an API. All the CANopen Manager functionality is implemented in the firmware, however the application must configure and control this firmware via the API.

The internal function modules *Network Manager*, *Configuration Manager* and *Process Image Manager/Scanner* provide the client application with encapsulated, convenient services for management of the complete CANopen network.

The following diagram shows the structure of the function modules of the CANopen Manager firmware:

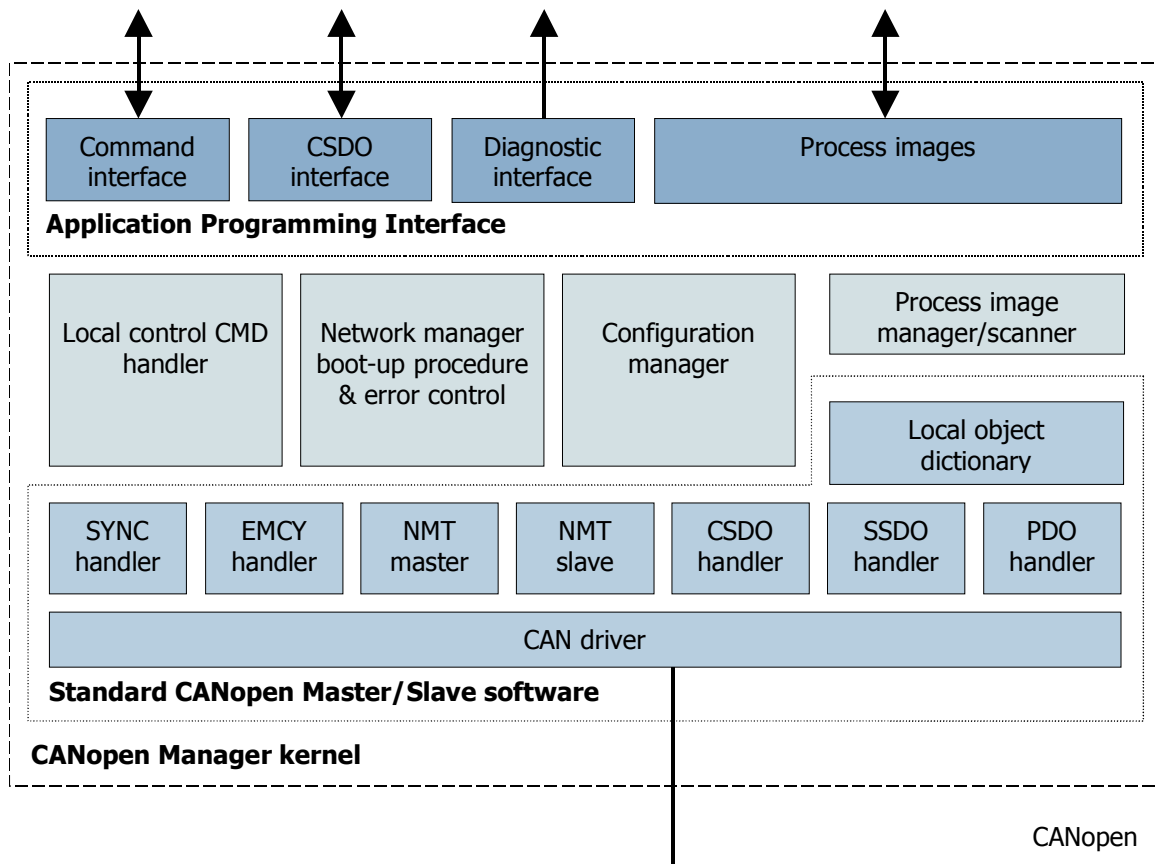


Figure 5-2: Software structure of the CANopen Manager

## 5.2 Services of the CANopen Manager

The kernel of the CANopen Manager is formed by a standard CANopen protocol stack. It is compliant to [1], which is responsible for executing the CANopen-specific commands. This stack has been extended such that the configurable automated boot-up procedure, a configurable network management, and the configuration manager are supported. The functionality of the extensions is described in more detail in CiA 302 [2]. In **Master Mode** the CANopen Manager supports both node guarding and the heartbeat mechanism. SDO block transfer is currently not supported by the CANopen Manager.

## 5.3 Boot-up Procedure

The boot-up procedure is described in CiA 302 and is used to transfer all slaves of a CANopen system into **Operational** state according to a defined procedure. A

differentiation is made between mandatory and optional slave modules, which have a different effect on the boot-up procedure.

The main features of the boot-up process are outlined below briefly. If the CANopen Manager is configured as the network master, the complete network with the exception of the CANopen Manager itself is reset and the **boot slave process** is started for each individual slave module. During a boot slave process the identity of a slave module is verified, if necessary its configuration is updated, monitoring of the module is started by the error control service, and the module is set to a pre-configured state. When the boot-up process is completed, all modules classified as mandatory should be in a defined state and the CANopen Manager can be started.

Start-up of the individual slave modules, of the network and of the CANopen Manager itself depends on the configuration of the **NMTStartup** object [1F82].

The CANopen Manager supports all boot-up process steps categorized as mandatory in accordance with CiA 302 [2].

### 5.4 Network Management

The network management of the CANopen Manager contains control and monitoring mechanisms for the complete network. The **error control service** monitors slave modules by means of **heartbeat** or **node guarding**. **Boot-up messages** and error control events of the individual modules are processed by the network management. Its reaction depends on the configuration of the **NMTStartup** [1F80] and **SlaveAssignment** objects [1F81].

In addition to the automated network management, the state (Operational, Pre-operational, Stopped) of the complete network or of an individual module can be controlled by the client application via the NMT-functions or by other modules on the network via the **RequestNMT** object [1F82]. It is thus possible to selectively activate or deactivate modules already during the boot-up process.

### 5.5 RequestNMT Object

The **RequestNMT** object [1F82] is an optional object dictionary entry of the CANopen Manager that is specified in CiA 302 [2]. As only the network master may execute NMT commands, the CANopen Manager supports the **RequestNMT** object, as in special cases slave modules or tools must be allowed to control the state of the network or of an individual module as well. With a write access (by means of an SDO) to the object, the required NMT service is requested and executed by the CANopen Manager. The state of a module can be requested by read access to **RequestNMT**.

## 5.6 Configuration Manager

During the boot-up process, the configuration manager configures all modules of the network based on their DCF object. As memory typically is limited, the CANopen Manager supports only the concise DCF format (object **ConciseDCF** [1F22]).

To reduce the time required for configuration, the CANopen Manager attempts to read the date and time of an already stored configuration of a module and compares these data with the expected values (objects [1F26] and [1F27]). If the entry read from the module is not equal to the expected value or if the queried module does not support the **VerifyConfiguration** object ([1020], see also section 5.8, Verify Configuration), complete configuration of the module is carried out. If the expected value and the read value are identical, configuration of the module is skipped.

## 5.7 Reset Configuration

To configure a module at run time, the CANopen Manager implements the optional **ConfigureSlave** object [1F25], which is specified in CiA 302. Via this object the **Request Configuration Service** can be called for a certain module or for all modules. Write access to the relevant sub index of this object is necessary to initiate this service.

## 5.8 Verify Configuration

The **VerifyConfiguration** object [1020] is categorized in CiA 301 as optional object. It is supported by the CANopen Manager. The object indicates the date and time of the current configuration of the CANopen Manager. It can be configured by a client application and used to identify its configuration before downloading a configuration to the CANopen Manager. In this way, unnecessary download of a configuration can be avoided.

## 5.9 Auto Configuration Mode

To simplify network management, the CANopen Manager provides an auto configuration mode in which the network is automatically scanned and a complete configuration of the network is created. This service is specific to the IXXAT CANopen Manager implementation and not described in CiA 302.

After a scan of the network by trying to inquire the device type [1000] for each possible node-ID, the CANopen Manager sets all identified modules to their default configuration (write access to **Restore default parameters** object

[1011sub01] and subsequent **NMT Reset Node** command). Then it reads out their **identity** objects [1018] and PDO configurations of all modules. By referring all valid PDOs to itself, the CANopen Manager creates a RPDO and TPDO table for the scanned network. The auto configuration mode requires the PDOs of the modules of the network to be configured such that no direct PDO communication takes place between the individual modules. Only in this way a non-ambiguous allocation between TPDOs and RPDOs is possible in the configuration of the CANopen Manager.

The auto configuration mode enables the client application to obtain an automatically generated overview of the configuration of the network from the information provided by the individual modules. This configuration contains information on the available modules stored in the local **SlaveAssignment** object [1F81] and in the other **Network List** objects [1F84]..[1F88]; the assignment of the process image and the allocation of the individual mapped objects and network variables to the individual modules. The automatically created configuration can then be further used as a basis for the specific configuration of the client application.

### 5.10 Initialization of the CANopen Manager

Initialization of the CANopen Manager can be requested by various services:

1. Via the command interface with the commands  
`CMM_InitFirmware()`  
`CMM_ResetNode()` for either all nodes or specifically the CANopen Manager  
`CMM_ResetComm()` for either all nodes or specifically the CANopen Manager
2. Via an external NMT command in *Slave Mode*  
**Reset Node** all nodes or specifically the CANopen Manager  
**Reset Communication** all nodes or specifically the CANopen Manager
3. Via external **RequestNMT** in *Master Mode*  
**Reset Node** all nodes or specifically the CANopen Manager  
**Reset Communication** all nodes or specifically the CANopen Manager
4. As the result of an error control event of a mandatory module whose configuration of the **NMTStartup** object requires the complete network to be reset by means of **Reset Node All Nodes**.

## 5.11 Object Dictionary default Values

The following table provides a partial overview of the default values of some entries in the communication profile section of the object dictionary.

Idx (hex)	Sub index	Name	Attrib	Obj Type	Data Type	Default Value
1000	00	Device Type	ro	VAR	UNSIGNED32	0x00000000
1001	00	Error Register	ro	VAR	UNSIGNED8	0x00
1005	00	COB-ID SYNC message	rw	VAR	UNSIGNED32	0x40000080
1018		Identity Object		RECORD		
	00	Number of entries	ro	VAR	UNSIGNED8	4
	01	Vendor ID	ro	VAR	UNSIGNED32	0x00000004
	02	Product code	ro	VAR	UNSIGNED32	0x01020134
	03	Revision number	ro	VAR	UNSIGNED32	0x00020303
	04	Serial Number	ro	VAR	UNSIGNED32	0x00000000

## 5.12 Special Manufacturer-specific Object Dictionary Entries of the CANopen Manager

The CANopen Manager implements manufacturer-specific object dictionary entries in the range [5F00] to [5FFF] as listed below.

Note: It is not possible to create dynamic object dictionary entries in this range!

Idx (hex)	Sub index	Name	Attrib	Obj Type	Data Type	Default Value
5F00		Status of the CANopen Manager		RECORD		
	00	Number of elements	ro	VAR	UNSIGNED8	3
	01	CANopen Manager event indication	ro	VAR	UNSIGNED16	
	02	CANopen Manager status	ro	VAR	UNSIGNED16	
	03	CANopen Manager communication status	ro	VAR	UNSIGNED8	
5F01		Assigned slaves bit list <sup>1</sup>		ARRAY		
	00	Number of elements	ro	VAR	UNSIGNED8	4
	01	Assigned slaves bit list: node-ID 1..32	ro	VAR	UNSIGNED32	
	02	Assigned slaves bit list: node-ID 33..64	ro	VAR	UNSIGNED32	

<sup>1</sup> Bit lists are organized such that bit 0 in sub index 1 corresponds to node-ID 1, therefore bit 31 in sub index 4 is not used.

## CANopen Manager Firmware

Idx (hex)	Sub index	Name	Attrib	Obj Type	Data Type	Default Value
	03	Assigned slaves bit list: node-ID 65..96	ro	VAR	UNSIGNED32	
	04	Assigned slaves bit list: node-ID 97..127	ro	VAR	UNSIGNED32	
5F02		Configured slaves bit list		ARRAY		
	00	Number of elements	ro	VAR	UNSIGNED8	4
	01	Configured slaves bit list: node-ID 1..32	ro	VAR	UNSIGNED32	
	02	Configured slaves bit list: node-ID 33..64	ro	VAR	UNSIGNED32	
	03	Configured slaves bit list: node-ID 65..96	ro	VAR	UNSIGNED32	
	04	Configured slaves bit list: node-ID 97..127	ro	VAR	UNSIGNED32	
5F03		Fault slaves bit list		ARRAY		
	00	Number of elements	ro	VAR	UNSIGNED8	4
	01	Fault slaves bit list: node-ID 1..32	ro	VAR	UNSIGNED32	
	02	Fault slaves bit list: node-ID 33..64	ro	VAR	UNSIGNED32	
	03	Fault slaves bit list: node-ID 65..96	ro	VAR	UNSIGNED32	
	04	Fault slaves bit list: node-ID 97..127	ro	VAR	UNSIGNED32	
5F04		Operational slaves bit list		ARRAY		
	00	Number of elements	ro	VAR	UNSIGNED8	4
	01	Operational slaves bit list: node-ID 1..32	ro	VAR	UNSIGNED32	
	02	Operational slaves bit list: node-ID 33..64	ro	VAR	UNSIGNED32	
	03	Operational slaves bit list: node-ID 65..96	ro	VAR	UNSIGNED32	
	04	Operational slaves bit list: node-ID 97..127	ro	VAR	UNSIGNED32	
5F05		Stopped slaves bit list		ARRAY		
	00	Number of elements	ro	VAR	UNSIGNED8	4
	01	Stopped slaves bit list: node-ID 1..32	ro	VAR	UNSIGNED32	
	02	Stopped slaves bit list: node-ID 33..64	ro	VAR	UNSIGNED32	
	03	Stopped slaves bit list: node-ID 65..96	ro	VAR	UNSIGNED32	
	04	Stopped slaves bit list: node-ID 97..127	ro	VAR	UNSIGNED32	
5F06		Preoperational slaves bit list		ARRAY		



Idx (hex)	Sub index	Name	Attrib	Obj Type	Data Type	Default Value
	00	Number of elements	ro	VAR	UNSIGNED8	4
	01	Preoperational slaves bit list: node-ID 1..32	ro	VAR	UNSIGNED32	
	02	Preoperational slaves bit list: node-ID 33..64	ro	VAR	UNSIGNED32	
	03	Preoperational slaves bit list: node-ID 65..96	ro	VAR	UNSIGNED32	
	04	Preoperational slaves bit list: node-ID 97..127	ro	VAR	UNSIGNED32	
5F07		Emergency slaves bit list		ARRAY		
	00	Number of elements	ro	VAR	UNSIGNED8	4
	01	Emergency slaves bit list: node-ID 1..32	ro	VAR	UNSIGNED32	
	02	Emergency slaves bit list: node-ID 33..64	ro	VAR	UNSIGNED32	
	03	Emergency slaves bit list: node-ID 65..96	ro	VAR	UNSIGNED32	
	04	Emergency slaves bit list: node-ID 97..127	ro	VAR	UNSIGNED32	
5F10		Node Error Count		ARRAY		
	00	Number of elements	ro	VAR	UNSIGNED8	127
	01	Received emergency messages of node-ID 1	ro	VAR	UNSIGNED8	
	..	..	..	..	..	
	7F	Received emergency messages of node-ID 127	ro	VAR	UNSIGNED8	
5F11	00	Generic Error Count	ro	VAR	UNSIGNED8	
5F12	00	Device Hardware Error Count	ro	VAR	UNSIGNED8	
5F13	00	Device Software Error Count	ro	VAR	UNSIGNED8	
5F14	00	Communication Error Count	ro	VAR	UNSIGNED8	
5F15	00	Protocol Error Count	ro	VAR	UNSIGNED8	
5F16	00	External Error Count	ro	VAR	UNSIGNED8	
5F17	00	Device-specific Error Count	ro	VAR	UNSIGNED8	
5F18		Nodes' Emergency History		ARRAY		
	00	Number of elements	ro	VAR	UNSIGNED8	127
	01	Emergency History of node-ID 1	ro	VAR	DOMAIN	
	..	..	..	..	..	
	7F	Emergency History of node-ID 127	ro	VAR	DOMAIN	

### 5.13 Configuration of the Run Time Behavior

A number of tasks are processed cyclically by the CANopen Manager. These include processing of the CAN receive queues, updates of the PI input, updates of the diagnostics buffer, transmission of PDOs, SDO handling, etc. This processing depends strongly on the run-time behavior of the CANopen Manager firmware, which can be configured via the *Task Time Configuration* object. This entry can be written in both slave and master mode using the local SDO access functions.

Object dictionary entry [5F80] Configuration of the run time behavior:

Idx (hex)	Sub index	Name	Attrib	Data Type	Value range	Default Value
5F80		Task Time Configuration				3
	00	Number of supported sub indices	ro	UNSIGNED8	3	3
	01	Task run time of the main task of the CANopen Manager (in ms)	rw	UNSIGNED8	2..10	2
	02	Maximum processing time of the high priority receive queue (in ms)	rw	UNSIGNED8	2..5	2
	03	Maximum processor time available to the main task of the CANopen Manager, given in per cent without decimal points	rw	UNSIGNED8	20..80	50



**These values should only be modified by experienced users. As long as the CANopen Manager is in auto configuration mode, this object entry should not be configured.**

## 5.14 Access to the local Object Dictionary

This section describes the restrictions or special features related to data consistency that apply when object entries managed by the CANopen Manager are accessed.

### 5.14.1 CiA 301 specific object entries

When the CANopen Manager is in either one of the *Network Initialization*, *Auto Configuration*, *Start Master Manager* or *Start Network* states, neither the PDOs nor the Consumer Heartbeat time nor the Producer Heartbeat time of the CANopen Manager can be reconfigured. These object entries are directly linked to both the configuration information contained in the concise Device Configuration Files and the slave assignment list. In the Auto Configuration state, these object entries can be read but they only reflect the network when scanning of the individual modules is completed and the CANopen Manager is in the sub-state *GETPI\_INFO*. Only from this point in time is the created configuration consistent.

### 5.14.2 CiA 302 specific object entries

All object entries related to the configuration of the network initialization **cannot** be configured by SDO in the states *Auto Configuration*, *Network Initialization*, *Start Master Manager* or *Start Network*.

This restriction concerns the following object entries:

- NMTStartup                                      Object [1F80]
- SlaveAssignment                                Object [1F81]
- All identity entries                             Object [1F84]..[1F88]
- BootTime                                        Object [1F89]
- ConciseDCF                                     Object [1F22]
- ConfigureSlave                                Object [1F25]
- ExpectedConfigurationDate                 Object [1F26]
- ExpectedConfigurationTime                 Object [1F27]

During auto configuration, these object entries can be read but their contents may be inconsistent if not all modules are scanned yet. The automatically created configuration is only consistent when the CANopen Manager is in the sub-state *GETPI\_INFO*.

The restrictions or special features that depend on the object entry are described in the following.

### **NMTStartup [1F80]**

The **NMTStartup** object cannot be configured in the *Slave Mode: Operational* state, as the CANopen Manager does not manage the network in the *Master Mode: Reset* state.

This version of the CANopen Manager does not support the **Flying Master** functionality described in [2]. Alteration of the **NMTStartup** object so that the flying master process would be supported (bit 5 = 1) is not accepted.



**After network initialization has been completed, the master functionality can be disabled again via the NMTStartup object.**

**This alteration only becomes effective when the configuration of the CANopen Manager has been saved and the CANopen Manager has been re-initialized.**

**After the initialization of the network the current network state is not retroactively changed by a reconfiguration of bits 4/6 of the NMTStartup object. These bits describe the reaction to an Error Control Event of a mandatory slave.**

### **SlaveAssignment [1F81]**

The current version of the CANopen Manager does not support bits 4 to 6 (**State Critical Node**, **Verify Software Version** and **Automatic Update**) of the **SlaveAssignment** object. A SlaveAssignment object that supports one of these services is not accepted.

After network initialization has been started, a reconfiguration of the SlaveAssignment of a module from “not assigned” slave or “optional” slave to “assigned and mandatory” slave is rejected if the module is not completely initialized and the configuration of the **NMTStartup** object stipulates a “Reset” or a “Stop” of the whole network with an Error Control Event of a “mandatory” module.

### **BootTime [1F89]**

The **BootTime** can also be configured in the state *BOOT\_END\_MISSING\_MAND* (see section 7.1.1, State of the CANopen Manager). Thus the timeout for missing mandatory modules can be extended (new BootTime = 0) or aborted (new BootTime != 0).

### ConciseDCF [1F22]

The individual concise DCFs share a common memory area. Thus the maximum size of a new concise DCF only depends on how much free memory is still available to it.

After a concise DCF has been successfully transmitted to the CANopen Manager, it is checked whether an older version of this concise DCF exists in the memory. If so, it is deleted and all subsequent concise DCFs are moved up. If transmission to the CANopen Manager failed, the transmitted data bytes are ignored and the configuration of the concise DCF is identical to the configuration before the failed download.

For this reason it is not possible to configure more than one entry of the concise DCF at the same time. If reading of a concise DCF is not yet completed, write access to a concise DCF is refused with a reference to the "Present Device State". If the configuration of a concise DCF is not yet completed, every read access to a concise DCF is refused with a reference to the "Present Device State". The download of a concise DCFs during a boot slave process is counted as a read access.

More than one concise DCF can be read out at the same time. However, a concise DCF cannot be read simultaneously by more than one SSDO.

### RequestNMT [1F82]

If the CANopen Manager is not configured as a master, **RequestNMT** cannot be read or written.

As long as the CANopen Manager does not have definite knowledge of the state of a slave module, the request for the state of a module returns the value "unknown". The state of a module can only be given after it has been added to the internal node list during its boot slave process.

If the CANopen Manager or the complete network is to be set to the **Stopped** state via RequestNMT, the RequestNMT command is not executed, as the CANopen Manager can then no longer communicate via SDO.

In the *Auto Configuration* state, only those RequestNMT commands are allowed that reset the CANopen Manager or the complete network.

In the *Network Initialization* state, start-up of the complete network is not allowed.

### Configure Slave [1F25]

The request to boot an individual module is complied with if the module to be booted is configured as a slave, is not **Operational** or is not currently being booted.

The request to boot the complete network is not complied with if even only one individual module is **Operational**.

### 5.15 Dynamically created Object Dictionary Entries

The user can create application specific entries in the object dictionary of the CANopen Manager. These entries are saved in the process images along with the network variables and can be mapped to PDOs.

The following parameters of the dynamically created object dictionary entry must be defined:

Parameter	Value range	Description
Object dictionary index	0x2000..0x5EFF 0x6000..0x9FFF	Index of the object dictionary entry
Object dictionary sub index	0x00..0xFF	Sub index of the object dictionary entry
Data type	INTEGER8 INTEGER16 INTEGER32 INTEGER64 UNSIGNED8 UNSIGNED16 UNSIGNED32 UNSIGNED64	These data types are supported for dynamically created object dictionary entries
Access type	rww/ro	Defines in which PI the variable is stored: <b>rww</b> variables are saved in the PI input and can be mapped to RPDOs <b>ro</b> variables are saved in the PI output and can be mapped to TPDOs
Mappable	yes/no	Defines whether the variable can be mapped to a PDO or not
Default value	any	Default value of the variables, the object dictionary entry is reset to this value with a reset command
PI offset	0..(size of the PI – variable size)	Byte offset, defines the position of the variables in the PI input or PI output

#### The following conditions are to be respected:

- Dynamic object dictionary entries can only be created in *Reset* state (see section 8.1.2, Master Mode: Reset) of the CANopen Manager!
- The user is responsible for observing CANopen conformity. For example, for every object dictionary index an entry with the sub index 0 must exist.

- Network variables and dynamically created object dictionary entries are saved in the same memory area. The user must ensure that there are no overlaps!
- After the dynamic object dictionary entries have been created, the command **STORE** must be executed in order to save the settings. Without this command the dynamic object dictionary entries would be reset, i.e. deleted, as soon as an **NMT Reset** is carried out.
- There is no command to delete a dynamically created object dictionary entry. All settings can only be reset and thus all dynamically created object dictionary entries deleted with the command **RESTORE** and a subsequent **NMT Reset** (see also section 5.16, Store/Restore).

### 5.16 Store/Restore

STORE is a write access to the object dictionary entry [1010sub1] of the CANopen Manager firmware, **RESTORE** is a write access to the object dictionary entry [1011sub1].

Please note that with the command **STORE** the settings are only saved in the RAM. After restarting the CANopen Manager firmware, i.e. after calling `CMM_InitBoard()` all settings must be made again.

The settings saved with **STORE** can be destroyed with the command **RESTORE**. With the next **NMT Reset** command, the default values of the settings are then restored.

### 5.17 Handshaking

For critical applications the handshake mechanism between the API-DLL and the CANopen Manager firmware can be activated. After the first handshake the CANopen Manager firmware starts its handshake timeout timer. If the handshake from the API-DLL is missing longer than the configured time then the firmware will execute several reactions.

The handshake interval and the timeout reactions of the CANopen Manager firmware are set with the function `CMM_InitFirmware()` (see section 11.1.4, `CMM_InitFirmware`). An handshake will be executed with call of `CMM_HandShake()` (see section 11.2.6, `CMM_HandShake`).

**!** After the first call of `CMM_HandShake()` the timeout timer of the CANopen Manager firmware starts. The application then must guarantee that the function `CMM_HandShake()` will be called cyclically within the handshake interval.

The meaning of the reactions (set with parameter `HSReaction` of `CMM_InitFirmware()`) are shown in the following:

Reaction	Meaning	valid in Master mode	valid in Slave mode	category
<code>CMM_HS_SEND_EMCY</code>	an Emergency message will be send with data field: 0x00, 0x62, 0x81, 0x00, 0x00, 0x00, 0x00, 0x00	yes	yes	EMCY
<code>CMM_HS_MASTER_NMT_PREOP_ALL</code>	NMT message Enter Pre-Operational all nodes will be sent	yes	no	NMT
<code>CMM_HS_MASTER_NMT_STOP_ALL</code>	NMT message Stop all nodes will be sent	yes	no	NMT
<code>CMM_HS_MASTER_NMT_RESETCOMM_ALL</code>	NMT message Reset Communication all nodes will be sent	yes	no	NMT
<code>CMM_HS_MASTER_NMT_RESETNODE_ALL</code>	NMT message Reset Node all nodes will be sent	yes	no	NMT
<code>CMM_HS_SLAVE_STOP_SENDING_HBT</code>	Sending of the own CANopen Heartbeat message is stopped	no	yes	HBT

The reactions can be combined using disjunction. From category NMT only one reaction can be selected. Invalid reactions – depending of the mode of the CANopen Manager firmware – will be ignored.



## 6 Structure of the Process Data Interface

### 6.1 Process Data Interface

The CANopen Manager and the application exchange process data via the process image. For the designation of the memory areas it is to be noted that the directions *input* and *output* refer to the point of view of the CMM-DLL, whereas the designation of the network variables refers to the point of view of the CANopen Manager.

The network variables [A000]..[A47F] are allocated to the PI output. The array of the network variables of the PI Input extends from [A480]..[A8FF]. The effectively available network variable array depends on the size of the available PI of the hardware.

The following diagram shows the data directions.

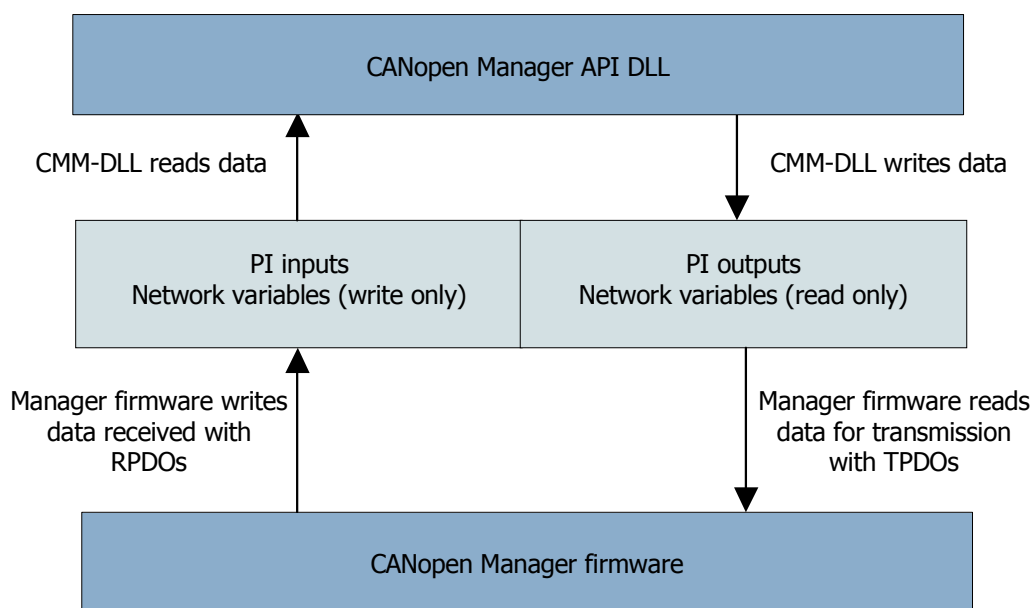


Figure 6-1: Data exchange of process variables via the PI

#### 6.1.1 Encoding rules

The data are stored in the process image specific to the processor (little Endian on the iPC-IXC16 PCI). All network variable types apart from **BOOLEAN** are supported.

The process image is allocated according to the principle of overlaid network variables (see also section 6.1.3, Overlaid Network Variables).


## Structure of the Process Data Interface

---

### 6.1.2 Data exchange between CMM-DLL and firmware

After the Manager Firmware has written new data to the PI input, it indicates this internally via a flag.

If a relevant callback function is defined via `CMM_DefineCallbacks()` or a user-defined Windows message is defined via `CMM_DefineMsgProcImg()`, the client application is informed directly of the change. For this purpose a poll thread is running inside the DLL, which inspects the flag at regular intervals. The standard duration of the interval is 4 ms, but can be set to within a millisecond with the function `CMM_SetInspeInterval()`.

 **Data received with synchronous RPDOs are copied into the process image with the next SYNC signal.**  
**As the run time of the main task of the CANopen Manager firmware is limited, only a limited number of RPDOs can be edited per cycle of the main task.**

When the CMM-DLL has written new data to the PI outputs, it also indicates internally with a flag that new data are to be transmitted.

The Manager Firmware then scans the complete valid PI outputs, i.e. the array of the PI outputs which is occupied with process variables mapped in TPDOs, for new data. Scanning is carried out with the aid of a copy of the PI outputs, by comparing each byte of the current outputs with its copy. Based on the offset of an altered byte, the associated TPDOs in which this object is mapped are determined and marked.

Transmission of the TPDOs according to their transmission type can not begin before the valid PI output is scanned completely. As an object could be mapped in several TPDOs, or a PDO could transmit data spread over the complete PI output array, a comparison of the complete valid PI output must be made before the TPDOs are transmitted. This is the only way to avoid multiple transmission of the same TPDOs.

### 6.1.3 Overlaid Network Variables

The network variables are held in one byte array each, separated according to PI inputs and PI outputs. The variables of one data direction are stored in the same physical memory irrespective of the data type. This means that the application is responsible for ensuring that there are no overlaps.

As the granularity of the PDOs and of the process image is 8 bits, the **BOOLEAN** data type is not supported.

**Supported network variables:**

Name	Data type	Start index of the block of a network variable type
<b>PI output</b> From the point of view of the application: output data  From the point of view of CANopen: input network variables  Maximum index range: [A000]..[A47F]	SIGNED8	[A000]
	SIGNED16	[A0C0]
	SIGNED24	[A140]
	SIGNED32	[A1C0]
	SIGNED40	[A2C0]
	SIGNED48	[A340]
	SIGNED56	[A3C0]
	SIGNED64	[A400]
	UNSIGNED8	[A040]
	UNSIGNED16	[A100]
	UNSIGNED24	[A180]
	UNSIGNED32	[A200]
	UNSIGNED40	[A280]
	UNSIGNED48	[A300]
	UNSIGNED56	[A380]
	UNSIGNED64	[A440]
	REAL32	[A240]
<b>PI input</b> From the point of view of the application: input data  From the point of view of CANopen: output network variables  Maximum index range: [A480]..[A8FF]	SIGNED8	[A480]
	SIGNED16	[A540]
	SIGNED24	[A5C0]
	SIGNED32	[A640]
	SIGNED40	[A740]
	SIGNED48	[A7C0]
	SIGNED56	[A840]
	SIGNED64	[A880]
	UNSIGNED8	[A4C0]
	UNSIGNED16	[A580]
	UNSIGNED24	[A600]
	UNSIGNED32	[A680]
	UNSIGNED40	[A700]
	UNSIGNED48	[A780]
	UNSIGNED56	[A800]
	UNSIGNED64	[A8C0]
	REAL32	[A6C0]

### 6.1.4 Default values

The process images are initialized to 0 on start-up of the firmware.

### 6.1.5 RPDO no queue

Each time the CANopen Manager receives an RPDO the number of the RPDO will be written into the RPDO no queue with a timestamp. Cyclically the PI Input will be updated with new received data. By that time an timestamp will be placed into the PI Input and also into the RPDO no queue (with a separator indication). So the user application knows what RPDO's belongs to a PI Input.

When using the RPDO no queue the application must call this function often enough to prevent queue overflows.

If the information about the received RPDO's is not needed, the application can ignore the RPDO no queue.

### 6.1.6 TriggerTPDO queue

Normally when new (changed) data are written into the Process Image Output the assigned TPDO's will be transmitted automatically.

Sometimes it is necessary to send an mapped object via TPDO (again) although the content of the object has not changed. The TriggerTPDO queue provides this possibility to trigger the assigned TPDO's manually.

To trigger the sending of all TPDO's that have mapped an specific object the function `CMM_TriggerPIOoffset()` is used.

## 7 Diagnostics Data

The diagnostics interface informs the application of the state of the CANopen Manager or of the network, the state of the individual modules and of the occurrence of certain errors or events such as failed modules or the faulty network state of a module. It serves as a basis for the decisions regarding network management.

The API-DLL can read out the diagnostics data both via the access functions of the diagnostics buffer `CMM_GetMasterStat()` and `CMM_GetSlavesStat()` and by local SDO access, as they are stored in the manufacturer-specific object dictionary entries [5F00]..[5F07].

If corresponding callback functions are defined by means of `CMM_DefineCallbacks()` or user-defined Windows messages have been given via `CMM_DefineMsgMaster()` / `CMM_DefineMsgSlave()`, the client application is informed directly of the altered state. For this purpose poll threads are running in the DLL, which inspect the diagnostics buffers at regular intervals. The standard duration of the interval is 4 ms, but can be set to within a millisecond with the function `CMM_SetInspectionInterval()`.

The contents of the diagnostics buffers is reset during initialization.

### 7.1 Status Information of the CANopen Manager

The state of the CANopen Manager can be read out by the client application and provides information on the configuration of the CANopen Manager, on the state of its communication with the network and which state its state machine is in.

#### API function:

```
CMM_GetMasterStat ( tCMM_HANDLE  hBoard
                   , WORD*        pMasterManagerState
                   , WORD*        fGlobalEvents
                   , WORD*        fConfigBits );
```

### 7.1.1 State of the CANopen Manager

The state of the CANopen Manager informs the application which state the internal state machine of the CANopen Manager is in. It is read only and can be retrieved both via the API function and by means of (local) SDO access.

#### Function parameters:

Lowbyte of \*pMasterManagerState

#### Object dictionary entry:

[5F00sub02] (Low byte)

#### Meaning of the states:

Value	Description
INIT (=0x00)	The CANopen Manager is not initialized. This corresponds to the <b>Initialization</b> state of a CANopen module. In this state the CANopen Manager cannot communicate with the network.
RESET (=0x40)	<i>Master Mode: Reset</i> The CANopen Manager is configured as a master in the <b>NMTStartup</b> object. The object dictionary of the CANopen Manager can be configured with SDOs via the CAN bus and the SDO command interface. The application can obtain read and write access to the object directories of all modules in the network via the SDO command interface. Network initialization and network management are not started yet.
<i>Slave Mode: The CANopen Manager is configured as a slave</i>	
SLAVE_STOPPED (=0x41)	The CANopen Manager is in the CANopen state <b>Stopped</b>
SLAVE_PREOP (=0x42)	The CANopen Manager is in the CANopen state <b>Pre-operational</b>
SLAVE_OP (=0x43)	The CANopen Manager is in the CANopen state <b>Operational</b>
<i>Master Mode: States of the network initialization</i>	
PREPARE_NET_INIT (=0x60)	Boot-up in accordance with CiA 302 [2]: The CANopen Manager carries out a check of the <b>SlaveAssignment</b> . Auto Configuration: The state machine of the "Auto Configuration" mode are reset.
NTW_RESET (=0x61)	The network is reset with NMT command <b>Reset Communication all Nodes</b> .

NTW_WAIT (=0x62)	The CANopen Manager waits for a definable time, so that the modules can carry out the <b>Reset Communication</b> command. In addition, in the <i>Auto Configuration</i> mode all structures that were configured during the network scan are rest.
BOOT_CON (=0x64)	The CANopen Manager carries out initialization of the individual modules in accordance with CiA 302
BOOT_AUTO (=0x65)	The CANopen Manager scans the individual modules and creates a configuration of the network and of the process image
GETPI_INFO (=0x66)	The application obtains information on the allocation of the process image after running the <i>Auto Configuration</i> mode
BOOT_END_MISSING_MAND (=0x70)	The network is scanned. At least one mandatory module is missing and the boot time has not yet expired
<p>The network is scanned.</p> <p>The high nibble of the state variable reflects the general network state (CLEAR, RUN, STOP, PREOPERATIONAL)</p> <p>The low nibble contains additional, more detailed information:</p> <ul style="list-style-type: none"> <li>▪ Bit 0: error bit for optional or unexpected modules <ul style="list-style-type: none"> <li>= 0: not an error</li> <li>= 1: at least one optional or unexpected module does not correspond to the expected network configuration</li> </ul> </li> <li>▪ Bit 1: error bit for mandatory modules <ul style="list-style-type: none"> <li>= 0: not an error</li> <li>= 1: at least one mandatory module does not correspond to the expectation</li> </ul> </li> <li>▪ Bit 2: general Operational bit: <ul style="list-style-type: none"> <li>= 0: no module is in the CANopen state <b>Operational</b></li> <li>= 1: at least one module is <b>Operational</b> (the CANopen Manager is not included)</li> </ul> </li> <li>▪ Bit 3: Operational bit of the CANopen Manager itself <ul style="list-style-type: none"> <li>= 0: the CANopen Manager is not <b>Operational</b></li> <li>= 1: the CANopen Manager is <b>Operational</b></li> </ul> </li> </ul>	
CLEAR (=0x8x)	The network is scanned. The command "Start CANopen Manager" or "Start network" is still missing.
RUN (=0xAx)	The network was set to the <b>Operational</b> state.
STOP (=0xC0)	The network was set to the <b>Stopped</b> state.
PREOPERATIONAL (=0xEx)	The network was set to the <b>Pre-operational</b> state.
FATAL_ERROR (=0x90)	A fatal error has occurred. The CANopen Manager must be re-initialized.

### 7.1.2 Communication state of the CANopen Manager

The communication state of the CANopen Manager records the state of the CAN controller and the state of the individual software queues, via which the CANopen Manager communicates with the network.

This status display is read only and can be read out both via the API function and by means of (local) SDO access.

#### Function parameters:

Highbyte of \*pMasterManagerState

#### Object dictionary entry:

[5F00sub02] (High byte)

#### Meaning of the communication display:

Bit	Description
Bit 0	= 1 an overrun of the low-priority receive queue occurred Via a low priority receive queue the CANopen Manager receives the Heartbeat and the Node Guarding messages as well as SSDOs and CSDOs.
Bit 1	= 1 an overrun of the CAN controller occurred
Bit 2	= 1 the CAN controller is Bus Off
Bit 3	= 1 the CAN controller has reached the Error status This bit is reset when the Error status is left again
Bit 4	= 1 the CAN controller has left the Error state again
Bit 5	= 1 an overrun of the low priority transmit queue occurred Via the low priority transmit queue the CANopen Manager transmits its Heartbeat, the Node Guarding requests, SSDOs and CSDOs.
Bit 6	= 1 an overrun of the high priority receive queue occurred Via the high priority receive queue the CANopen Manager receives RPDOs, NMT commands, the Sync message and emergency messages.
Bit 7	= 1: an overrun of the high priority transmit queue occurred. Via the high priority transmit queue the CANopen Manager transmits TPDOs, NMT commands, the Sync message and its emergency message



## 7.1.3 Event Indication

The event indication informs the client application of the occurrence of certain events and errors.

### Function parameter:

\*fGlobalEvents

### Object dictionary entry:

[5F00sub01]

### Description of individual bits:

Bit / name	Cause	Effect
Bit 0 = 1 FATE	This bit is always set when <ul style="list-style-type: none"> <li>▪ an error has occurred in the communication with the network. The communication state of the CANopen Manager indicates the exact reason.</li> <li>▪ Local software error of the CANopen Manager.</li> </ul>	The CANopen Manager is in the state <i>Fatal Error</i> .
Bit 1 = 1 NIDE	Only valid in <i>Master Mode</i> . A module uses the node number of the CANopen Manager.	The CANopen Manager is in the state <i>Fatal Error</i> .
Bit 2 = 1 MSE	Only valid in <i>Master Mode</i> . <b>Error control event</b> of a mandatory module.	The reaction to this event depends on the configuration of the <b>NMTStartup</b> object. If the application reserves the decision for itself, it must react! This bit is only relevant if the configuration of the <b>NMTStartup</b> object does not stipulate a reset of the complete network including the CANopen Manager. In this case a reset is carried out without the application being notified beforehand.
Bit 3 = 1 MNCE	Only valid in <i>Master Mode</i> . Identity error or faulty concise DCF of a mandatory module.	The CANopen Manager is in the state <i>Fatal Error</i> .
Bit 4 = 1 OIE	Only valid in <i>Master Mode</i> . Identity error of an optional module.	The module involved is set to the state <b>Stopped</b>

## Diagnostics Data

Bit 5 = 1 PIE	Only valid in <i>Master Mode</i> . Creation of a configuration, of the process image and of the PDOs failed during auto configuration mode.	The CANopen Manager is in the state <i>Fatal Error</i> .
Bit 6 = 1 ACE	Only <i>Master Mode</i> . during scanning of the network in auto configuration mode. <ul style="list-style-type: none"> <li>▪ an <b>error control event</b> of an already scanned module occurred</li> <li>▪ a module registered late on the network with its boot-up message</li> </ul>	The CANopen Manager is in the state <i>Fatal Error</i> .
Bit 7 = 1 NMTE	This bit is always set when a bit in one of the bit lists changes.	
Bit 8 = 1 ASE	Only valid in <i>Master Mode</i> . At the beginning of the boot-up procedure the CANopen Manager checks the individual <b>SlaveAssignment</b> objects [1F81]. This bit is set when the <b>SlaveAssignment</b> of a module contains features that are not supported by the CANopen Manager (e.g. bit 4 to bit 6 of object [1F81])	The CANopen Manager is in the state <i>Fatal Error</i> .
Bit 9 = 1 PDOLEN_ERR	The CANopen Manager has received an RPDO with too few data bytes.	The CANopen Manager is in the state <i>Fatal Error</i> .
Bit 10 = 1 CONFIG_ERR	Only valid in <i>Master Mode</i> . A concise DCF is faulty in itself or does not match the object dictionary of the slave module.	The CANopen Manager is in the state <i>Fatal Error</i> .
Bit 11 API_ROVR	This bit indicates an queue overrun of the CSDO interface.	The application decides on the consequences
Bit 12 = 1	Reserved	
Bit 13 = 1	Reserved	
Bit 14 = 1 RSCN	Only valid in <i>Master Mode</i> . the state of the complete network was altered by <b>RequestNMT</b> object.	
Bit 15 = 1 RSCM	Only valid in <i>Master Mode</i> . The state of an individual module was altered by <b>RequestNMT</b> object.	



**The CANopen Manager sets the complete network to the state Stopped if it is configured as a master and assumes the state Fatal Error.**

### 7.1.4 Configuration of the CANopen Manager

The configuration informs the application in a simple way of the most important settings of the CANopen Manager:

- the configuration of the **NMTStartup** object
- whether it participates in the synchronization mechanism
- whether it is configured as a **SYNC** producer or consumer

As the configuration of the **NMTStartup** object can be altered during run time, it is important, especially in view of the master functionality and the reaction auf to a mandatory Error Control event, that the application is informed of the basic configuration of the **NMTStartup** object.

For treatment of the process data, it is important that the application knows whether the CANopen Manager participates in the synchronization mechanism or not.

#### Function parameter:

\*fConfigBits

#### Object dictionary entry:

none

#### Description of the individual bits:

Bit	Description
Bit 0 – 5 comprise the configuration of the <b>NMTStartup</b> object: Bit 0 – 3 correspond here to the bits 0-3 of the <b>NMTStartup</b> object [1F80H]	
Bit 0	Master/slave bit: = 0 the CANopen Manager is configured as a slave = 1 the CANopen Manager is configured as a master
The following bits describe the start-up behavior of the CANopen Manager during the boot-up procedure:	
Bit 1	Start mode of the slaves: = 0: The slave modules are started individually during the boot slave process = 1: The CANopen Manager starts the network at the end of the boot-up procedure Bit 1 is only valid if bit 3 = 0!
Bit 2	Self-start permission: = 0: The CANopen Manager may start itself. This also applies to the slave mode! = 1: the CANopen Manager may not start itself
Bit 3	General start permission: = 0: CANopen Manager may set the slave modules to operational = 1: The application starts the slave modules Bit 3 is only valid if bit 0 = 1!

## Diagnostics Data

---

The following bit indicates who must react to an Error Control event of a mandatory module:

Bit 4                      Reaction permission:  
= 0:    The application reserves the right to react  
= 1:    CANopen Manager acts. The reaction is configured in the **NMTStartup** object

Bit 5                      Reserved: always 0

Synchronization mechanism

Bit 6                      Synchronization bit:  
= 0:    the value of **Communication Cycle Period** (Index 1006H) is 0  
= 1:    the value of **Communication Cycle Period** (Index 1006H) is not equal to 0

Bit 7                      Sync consumer/producer bit:  
= 0:    the CANopen Manager is configured as a **SYNC** consumer  
= 1:    the CANopen Manager is configured as a **SYNC** producer

Bit 8 – bit 15          free

## 7.2 Slave Diagnostics

### 7.2.1 Overview

The slave diagnostics informs the API-DLL of the state of the individual modules in the network. For the slave diagnostics, it is necessary to distinguish between the *Master Mode* and the *Slave Mode*.

In *Master Mode* the state of all modules, i.e. node numbers 1 – 127 (including the CANopen Manager) is displayed after the boot-up procedure or the auto configuration mode is started.

In *Slave Mode* or in the state *Master Mode: Reset*, only the states of those modules are logged that are entered in the **consumer heartbeat list** of the CANopen Manager and whose consumer heartbeat time is not equal to 0.

As the state of an individual module can be requested by **RequestNMT** (read access to object [1F82]), the slave diagnostics is not created according to nodes but according to states. Every possible state receives its own bit list `tCMM_SLAVEFLAGS`, in which a module can be configured according to its state.

The CANopen Manager differentiates between the following states:

- Assigned slaves                      → **fAssigned** bit list
- Configured slaves                    → **fConfigured** bit list
- Preoperational slaves               → **fPreoperational** bit list
- Stopped slaves                       → **fStopped** bit list
- Operational slaves                  → **fOperational** bit list
- Configuration error                 → **fMismatch** bit list

- Module internal error → **fEmergency** bit list

Each node number is allocated a certain bit within the bit list:

- Node number  $i \leftrightarrow$  bit  $(i - 1)$

The slave diagnostics data are read only and can be read out both via the API-function and by means of (local) SDO access.

### API function:

```
CMM_GetSlavesStat ( tCMM_HANDLE          hBoard
                   , tCMM_SLAVEFLAGS*    fAssigned
                   , tCMM_SLAVEFLAGS*    fConfigured
                   , tCMM_SLAVEFLAGS*    fMismatch
                   , tCMM_SLAVEFLAGS*    fEmergency
                   , tCMM_SLAVEFLAGS*    fOperational
                   , tCMM_SLAVEFLAGS*    fStopped
                   , tCMM_SLAVEFLAGS*    fPreOperational );
```

If the CANopen Manager is configured as a slave or if it is in the state *Master Mode: Reset*, the following applies to the **fAssigned**, **fPreOperational**, **fStopped**, **fOperational** and **fMismatch** bit lists:

- In the **fAssigned** bit list, those modules are marked which are entered in the consumer heartbeat list of the CANopen Manager and whose consumer heartbeat time is not equal to 0.
- Only those modules are logged which are marked in the **fAssigned** bit list.
- The bits of the modules which are not logged are set to 0.
- If the state of a module is no longer to be displayed (e.g. if it was deleted from the consumer heartbeat list or its consumer heartbeat time was set to 0), its bits are reset in the above-mentioned bit lists.
- The **fConfigured** bit list and the **fEmergency** bit list are not supported.

### 7.2.2 Structure of the bit lists

In the following, the allocation of the node numbers to the entries of the bit lists is shown in more detail. This allocation is the same for all bit lists of the slave diagnostics.

#### Byte – bit allocation:

## Diagnostics Data

Byte	Description
Byte 0 Bit number: 0 - 7	Node number 1 – 8: Bit 0 ↔ node number 1 Bit 7 ↔ node number 8
Byte 1 Bit number: 8 - 15	Node numbers 9 – 16: LSBit ↔ node number 9 MSBit ↔ node number 16
...	...
Byte 15 Bit number: 120 - 127	Node numbers 121 – 127: LSBit ↔ node number 121 MSBit ↔ node number 128: free

If the bit list information is accessed via SDO, the following allocation pattern is used:

Sub-index	Data type	Value	Attribute
0	UNSIGNED8	4	RO
1	UNSIGNED32	Node number 1 – 32	RO
2	UNSIGNED32	Node number 33 – 64	RO
3	UNSIGNED32	Node number 65 – 96	RO
4	UNSIGNED32	Node number 97 – 127	RO

The meaning of the bit states does not change if a bit list is accessed via SDO.

### 7.2.3 Bit list assigned slaves

In *Master Mode*, except for the state *Master Mode: Reset*, those modules are marked in the assigned slaves bit list which are configured as slaves in the slave assignment. If a module is no longer configured as a slave, the corresponding bit is also reset in the `fAssigned` bit list.

In *Slave Mode* and in the state *Master Mode: Reset*, the states of the modules entered in the consumer Heartbeat list are also logged.

#### Function parameter:

\*`fAssigned`

#### Object dictionary entry:

Object: [5F01]

### 7.2.4 Bit list configured slaves

After a module that is allocated to the master as a slave module in the slave assignment, has been successfully configured during the boot-up procedure, it is marked accordingly in the **fConfigured** bit list.

A module is taken out of this list again when it is no longer allocated to the master as a slave module, if an error control event of the module occurred or the module was reset via NMT command.

This bit list is not supported in *Slave Mode*.

**Function parameter:**

\*fConfigured

**Object dictionary entry:**

Object: [5F02]

### 7.2.5 Bit list configuration error

If a module has a state that does not match its expected state, it is marked in the **fMismatch** bit list. In addition, a module is marked as faulty if it is allocated to the master as a slave but is not (yet) not initialized or its concise DCF is faulty. A module is also marked as faulty if it is not allocated to the master as a slave but is still in the network.

**Function parameter:**

\*fMismatch

**Object dictionary entry:**

Object: [5F03]

**Description of configuration errors:**

Error	Cause	Effect
A module is not allocated to the master as a slave but is present in the system		
The module uses the node numbers of the CANopen Manager	<ul style="list-style-type: none"> <li>▪ The module was detected during network initialization</li> <li>▪ It registered on the network with its boot-up message</li> </ul>	The <b>fMismatch</b> bit list is deleted. Only this module is still marked. The NIDE bit of the "Event Indication" indicates this error. The CANopen Manager is set to the state <i>Fatal Error</i> .

## Diagnosics Data

The module uses a free node number	<ul style="list-style-type: none"> <li>▪ The module was detected during network initialization</li> <li>▪ It registered on the network with its boot-up message</li> <li>▪ The module was allocated to the master as a slave and was present. However, the slave configuration was subsequently altered.</li> </ul>	The search algorithm checks cyclically whether the module is still in the network.
A module is allocated to the master as a slave		
Incorrect slave assignment	<ul style="list-style-type: none"> <li>▪ At the beginning of the boot-up procedure, the CANopen Manager checks the individual slave assignments. The CANopen Manager then detects a slave assignment that configures features of a slave which are not supported by the CANopen Manager (e.g. bit 4 to bit 6 of the object 1F81H).</li> </ul>	<p>The check of the other slave assignments is no longer carried out. Only this module is marked as faulty. The ASE bit of the "Event Indication" indicates this error. The CANopen Manager is set to the state <i>Fatal Error</i>.</p>
The module is missing	<ul style="list-style-type: none"> <li>▪ The module has not registered during the boot slave process.</li> <li>▪ During its boot slave process an SDO timeout occurred: the module is reset via NMT command.</li> <li>▪ An error control event of the module occurred. The module was reset by NMT command.</li> <li>▪ The module was reset by NMT-function or by <b>RequestNMT</b> object.</li> </ul>	<p>The search algorithm checks cyclically whether the module is still in the network. The module is marked in the following bit lists with the value 0: <b>fPreoperational-</b>, <b>fStopped-</b>, <b>fOperational-</b> and <b>fConfigured bit list</b></p>
Identity error	<ul style="list-style-type: none"> <li>▪ During the boot slave process, the identity of the module is checked. It is a different module than expected.</li> </ul>	<p>Mandatory module: The <b>fMismatch</b> bit list is deleted. Only the faulty module is still marked. The NMCE bit of the "Event Indication" indicates this error. The CANopen Manager is set to the state <i>Fatal Error</i>.</p> <p>Optional module: The module is set to the state <b>Stopped</b>. The error is indicated via the OIE bit of the Event Indication.</p>
Concise DCF error	<p>When downloading the concise DCF, the CANopen Manager detects that the concise DCF of this module is faulty:</p> <ul style="list-style-type: none"> <li>▪ Length of the concise DCF and the number of entries do not match.</li> <li>▪ The concise DCF contains at least one entry that is not supported by the target node.</li> </ul>	<p>The <b>fMismatch</b> bit list is deleted. Only the module concerned is still marked. The CONFIG_ERR Bit of the Event Indication indicates this error. If the module is additionally configured as mandatory, the MNCE bit is also set. The CANopen Manager is set to the state <i>Fatal Error</i>.</p>



### Specific auto configuration mode error

SDO timeout	<ul style="list-style-type: none"> <li>▪ SDO timeout when reading out the object dictionary of a module during the <i>Scan slave process</i>.</li> </ul>	The <b>fMismatch</b> bit list is deleted. Only the module that caused the error is still marked.
Error control event	<ul style="list-style-type: none"> <li>▪ Error control event of an already scanning module.</li> </ul>	The ACE bit of the Event Indication indicates this error.
Boot-up message	<ul style="list-style-type: none"> <li>▪ A module registers out of sequence.</li> </ul>	The CANopen Manager is set to the state <i>Fatal Error</i> .
COB ID allocation	<ul style="list-style-type: none"> <li>▪ The PDO configuration of the auto configuration mode requires that there be exactly one receiver and one transmitter for each PDO. In this error case, an identifier is used more than once.</li> </ul>	The <b>fMismatch</b> bit list is deleted. Only the module that caused the error is still marked. The PIE bit of the Event Indication indicates this error. The CANopen Manager is set to the state <i>Fatal Error</i> .
Concise DCF	<ul style="list-style-type: none"> <li>▪ The configured size of the concise DCF buffer is insufficient.</li> </ul>	
PDO	<ul style="list-style-type: none"> <li>▪ The number of defined PDOs is insufficient.</li> </ul>	
Process image	<ul style="list-style-type: none"> <li>▪ The size of the process image is too small.</li> </ul>	
Network variables	<ul style="list-style-type: none"> <li>▪ The defined array of the network variables is too small</li> </ul>	
Allocation error	<ul style="list-style-type: none"> <li>▪ The allocation of the process image is faulty. This error was detected during execution of the process image information commands <i>GETPI_INFO</i>.</li> </ul>	

### General errors

Local software error of the CANopen Manager	<ul style="list-style-type: none"> <li>▪ A state machine of the CANopen Manager goes into an undefined state.</li> <li>▪ A variable which the CANopen Manager manages itself has an invalid value.</li> </ul>	<p>The <b>fMismatch</b> bit list is deleted. Only the CANopen Manager itself is marked.</p> <p>The CANopen Manager is set to the state <i>Fatal Error</i>.</p> <p>If this error occurs in the state <i>BOOT_AUTO</i>, the ACE bit is set in the "Event Indication", otherwise the FATE bit</p>
---	---	--

### 7.2.6 Bit list operational slaves

This bit list indicates which modules are in the **Operational** state. If the state of the module changes, the corresponding bit is reset.



Particular attention must be paid to modules which are configured as a slave and are not monitored by the error control service. If the state of such a module changes, this is not noticed by the CANopen Manager unless the module registers on the network with its boot-up message.

**Function parameter:**

\*fOperational

**Object dictionary entry:**

Object: [5F04]

### 7.2.7 Bit list stopped slaves

This bit list indicates which modules are in the **Stopped** state. If the state of a module changes, the corresponding bit in the bit list is reset.



Particular attention must be paid to modules which are configured as a slave and are not monitored by the error control service. If the state of such a module changes, this is not noticed by the CANopen Manager unless the module registers on the network with its boot-up message. If the expected state of the module is Stopped, it is set to this state by the master and marked accordingly in the stopped bit list of the diagnostics interface.

**Function parameter:**

\*fStopped

**Object dictionary entry:**

Object: [5F05]

### 7.2.8 Bit list preoperational slaves

This bit list indicates which modules are in the **Pre-operational** state. If the state of a module changes, the corresponding bit in the bit list is reset.



**Particular attention must be paid to modules which are configured as a slave and are not monitored by the error control service. If the state of such a module changes, this is not noticed by the CANopen Manager unless the module registers on the network with its boot-up message. Then it is first marked as Pre-operational in the bit list. If the expected state of the module is Stopped, it is set to this state by the master and the corresponding bit lists of the diagnostics interface are adapted. Otherwise the module remains in the Pre-operational state until it is expressly set to another state.**

**If a module is not marked in the Pre-operational, Operational or in the Stopped bit list, it is assumed that the module is not present in the network or its state is unknown.**

**Function parameter:**

\*fPreOperational

**Object dictionary entry:**

Object: [5F06]

### 7.2.9 Bit list module internal errors

In the *Master Mode* the CANopen Manager analyses the received emergency messages. If a module signals that it is not error-free, its bit is set in the fEmergency bit list. Likewise it is reset if the module indicates error-free. The CANopen Manager itself is not included in this list. At the beginning all modules are marked as error-free (default value).



**Only after the module has been added to the internal node list of the CANopen Manager during the boot slave process are its emergency messages received.**

**Function parameter:**

\*fEmergency

**Object dictionary entry:**

Object: [5F07]

### 7.3 Emergency Statistic and History

The emergency statistic and history is only supported in *Master Mode*. Its entries can only be read by SDO. The emergency statistic records the frequency of selected error types in the network (error code-specific error counter). In addition, the received emergency messages of each individual module are counted (node error count). A separate emergency queue is set up in the emergency history for each node number, so that the error history of the last **four** received emergency messages of a module are hold. The CANopen Manager itself is not recorded in the emergency statistic. Its emergency messages are not collected in the emergency history.

#### 7.3.1 Node Error Count

The CANopen Manager counts the received emergency messages of each node. The error-free messages are also counted. If the counter limit of 255 is reached, the counter is not incremented further.

#### Object dictionary entry:

Object: [5F10]

#### Description of the object entry:

Sub-index	Data type	Value	Attribute
0	UNISIGNED8	127	RO
1	UNISIGNED8	Number of received emergency messages of node number 1	RO
...			
127	UNISIGNED8	Number of received emergency messages of node number 127	RO

#### 7.3.2 Error code-specific error counter

The CANopen Manager counts and sorts the received emergency messages according to the main error types.

If a counter has reached the value 255, it is not incremented further.

#### List of recorded error types:

Index of the object entry	Name	Error Code
[5F11]	Generic Error Count	10xxH
[5F12]	Device Hardware Error Count	50xxH
[5F13]	Device Software Error Count	60xxH

[5F14]	Communication Error Count	81xxH
[5F15]	Protocol Error Count	82xxH
[5F16]	External Error Count	90xxH
[5F17]	Device specific	FFxxH

### Description of an individual object entry:

Sub-Index	Data type	Value	Attribute
0	UNSIGNED8	Number of received emergency messages of this error type	RO

### 7.3.3 Emergency history

The received emergency messages of an individual module are chronologically logged in a separate emergency queue.

This emergency queue is designed as a FIFO buffer. The last four received emergency messages of each module are hold. When the queue is full, the emergency message last received overwrites the oldest entry.

#### Object dictionary entry:

Object: [5F18]

#### Description of the object entry:

Sub-index	Data type	Value	Attribute
0	UNSIGNED8	127	ro
1	DOMAIN	Emergency history of node number 1	ro
...			
127	DOMAIN	Emergency history of node number 127	ro

#### Description of the domain:

Byte	Meaning
0	Number of the emergency message last received
1 – 8	Emergency message position 1
9 – 16	Emergency message position 2
17 – 24	Emergency message position 3
25 – 32	Emergency message position 4

#### Note:

- If no emergency message has yet been received by a module, byte 0 returns the value zero. The size of the domain is one.

## Diagnosics Data

---

With each received emergency message the domain raises at 8 bytes. When the maximum of 33 byte is reached, then the older emergency messages are overwritten by the newer ones. So always the last four received emergency messages can be read out via SDO.

The value of byte 0 gives back the position of the last received emergency message.

- The data of the emergency messages are copied to the queue in the same order in which they were received (LSB first). Each emergency message consists of exactly 8 bytes.

### Example of a domain entry:

In the following example there were received more than four emergency messages. The domain has reached its maximum of 33 bytes. In byte 0 the value is 2. So the last received emergency message can be read out at byte 9. The emergency message before the last one stands at position 1, etc.

Byte	Value	Attribute
0	2	The emergency message 2 is the last received
1 – 8	Emergency message position 1	Penultimate emergency message
9 – 16	Emergency message position 2	Emergency message last received
17 - 24	Emergency message position 3	Oldest emergency message of this history
25 - 32	Emergency message position 4	This message is newer than the previous one



**A read access to the emergency history of the CANopen Manager is answered with the abort code: 0800 0021H.**

## 7.4 Default Values

After power-on, the data of the diagnostics interface are completely deleted and initialized with 0. This process is also carried out with each initialization of the CANopen Manager.

If the initialization of the CANopen Manager was successful, the slave diagnostics indicates the state of the modules that were also logged by the consumer heartbeat log.

## 8 States of the CANopen Manager

The CANopen Manager is implemented as a state machine, which is illustrated in the diagram below:

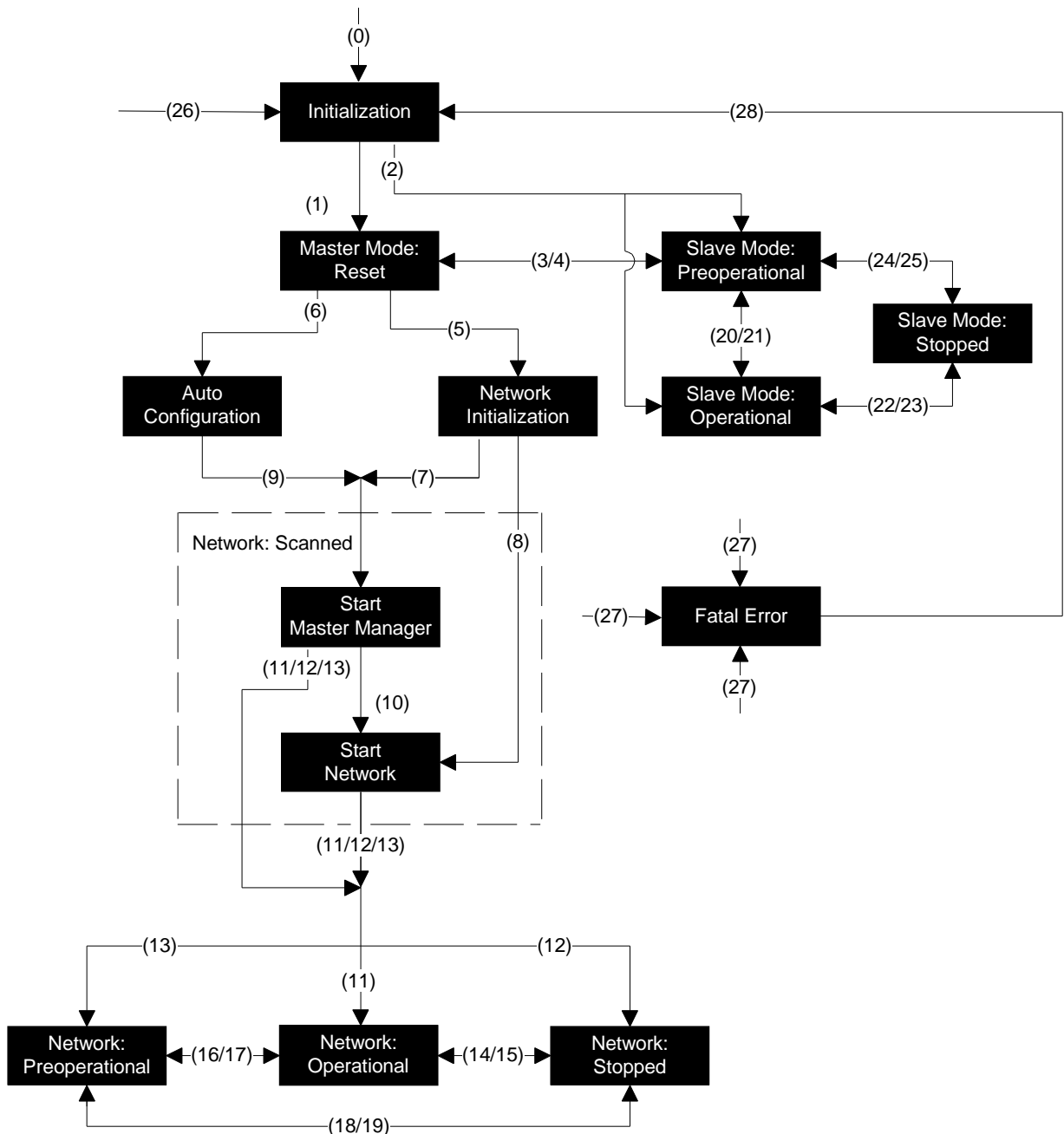


Figure 8-1 State diagram of the CANopen Manager

### 8.1.1 Initialization

This state is automatically assumed after power-on and corresponds to the **Initialization** state of a CANopen module. Initialization of the CANopen Manager can be requested from every state of the CANopen Manager.

The CANopen Manager cannot communicate with the network in this state. Access to the object dictionary of the CANopen Manager is not possible.

### 8.1.2 Master Mode: Reset

To reach this state, the CANopen Manager must be configured as a master in the **NMTStartup** object [1F80].

The object dictionary of the CANopen Manager can be configured both by SDOs via the CAN bus and directly via the local SDO access functions. In this state the client application can also have read and write access to the object directories of all modules via the SDO functions.

Although the network initialization and the network management are not yet started, the diagnostics interface nevertheless provides information on the state of the modules that are entered in the consumer Heartbeat list of the CANopen Manager.

In *Master Mode: Reset* state the CANopen Manager transmits heartbeat messages. It is also possible to create dynamic object dictionary entries in this state.



**All NMT commands, except Start Remote Node, are supported. Their execution can be requested via the command interface or by means of RequestNMT object via the CAN bus.**

**The CANopen Manager is either pre-operational – it automatically assumes this state after successful initialization – or stopped. The stopped state can be requested via the corresponding NMT function.**

### 8.1.3 Network Initialization

In this state the CANopen Manager runs the initialization of the network conform to CiA 302 [2].

The network initialization state is divided into the following sub-states, which are run through in direct succession:



Sub-state	Description
PREPARE_NET_INIT ( = 0x60)	<p>The CANopen Manager carries out a check of the <b>SlaveAssignment</b>. If it detects a configuration that is not supported, the network initialization is aborted.</p> <p>All modules that are configured as an assigned slave are marked in the diagnostic interface as assigned slave.</p>
NTW_RESET ( = 0x61)	<p>The network is reset by NMT command <b>Reset Communication (all nodes)</b>.</p>
NTW_WAIT ( = 0x62)	<p>The CANopen Manager waits for a configurable time, so that the modules can carry out the <b>Reset Communication</b> command.</p> <p>After this timeout has expired, the CANopen Manager begins to scan the network.</p> <p>This timeout and the SDO timeout should not be selected too short, in order to prevent a transmit queue overrun when scanning the network.</p>

## States of the CANopen Manager

---

BOOT\_CONF  
( = 0x63)

The CANopen Manager scans the complete node number array from 1 to 127 to ensure that no unexpected modules are present in the network.  
The CANopen Manager scans and configures one module after the other. It always begins with node number 1.

When a module is present

- the CANopen Manager checks whether it is configured as a slave.

The module is configured as a slave:

- Its identity is checked
- The configuration manager checks whether the module has to be configured and if so configures it
- The module is marked as correctly configured in the diagnostics interface
- The module is added to an internal node list and from now on the emergency and boot-up messages can be received
- The relevant error control service is started: Heartbeat or node guarding
- It is checked whether the module is to be started individually. If so, it is started. Otherwise the module is set to the expected state. The expected state is stipulated by the configuration of **NMTStartup**, but can be altered by NMT functions or by **RequestNMT** object.

The module is not configured as a slave:

- The *scan endlessly* mechanism checks cyclically whether the module is still present. The module is marked as faulty in the diagnostics interface.

If a module is not present:

- The module is added to the internal node list, so that emergency and boot-up messages can be received.

The module is configured as a slave:

- The *scan endlessly* mechanism checks cyclically whether the module is present in the network. The module is marked as faulty in the diagnostics interface.

No modules are booted in parallel.

If a module does not present itself as expected on the basis of the configuration, it is marked as faulty in the diagnostics interface.

---

BOOT\_END\_MISSING\_  
MAND  
( = 0x70)

When the network is scanned and not all mandatory modules have been correctly initialized, the CANopen Manager checks whether the boot time has expired.

As long as this boot time has not expired and not all mandatory modules are booted, the CANopen Manager remains in this state.

As soon as all mandatory modules are initialized or the boot time has expired, the display of the network state is updated.

---



If during the initial network initialization process a slave module was missing but has been detected again, the boot slave process is continued as normal. Then this initially missing module is booted. If more than one module is to be booted subsequently, the CANopen Manager begins with the as not yet configured mandatory modules. Only after this the optional slave modules are booted. Only after all found modules are booted the CANopen Manager continues with the regular network initialization.



If the modules are to be started individually by the CANopen Manager at the end of the boot slave process, the CANopen Manager checks whether all mandatory modules are booted. The configured modules are only started when all mandatory modules are configured.

This check is not performed if an individual module is set to operational by means of a command of the command interface or of the RequestNMT object.

### Restrictions:

All object entries of the CANopen Manager which concern the configuration of the network initialization cannot be reconfigured during initialization process. This applies to the following objects:

- NMTStartup   Object [1F80]
- Consumer heartbeat time:                                 Object [1016]
- SlaveAssignment   Object [1F81]
- Identity objects:   Objects [1F84]..[1F88]
- Concise DCF   Object [1F22]
- Configure slave   Object [1F25]
- Expected configuration date/time:                        Objects [1F26] and [1F27]
- BootTime:    Object [1F89]  
    The boot time can be configured in the sub-state  
    *BOOT\_END\_MISSING\_MAND* in order to end or prolong this state.

In the *Network Initialization* state, read access of the object dictionary of the CANopen Manager is possible directly via the local SDO access functions and the object directories of the slave modules via the SDO functions.

## States of the CANopen Manager

---

Only the states of individual slave modules can be altered by the NMT functions or the **RequestNMT** object. The only exceptions are **Reset Communication** or **Reset Node**, which can alter the state of the complete network including the CANopen Manager itself.

### 8.1.4 Auto Configuration

In the *Auto Configuration* state, the CANopen Manager scans the network and independently creates a network configuration with the information collected. The individual modules are set to their default configuration and then scanned.

This state is divided into the following sub-states:

Sub-state	Description
PREPARE_NET_INIT ( = 0x60)	The CANopen Manager resets the state machine of the auto configuration mode.
NTW_RESET ( = 0x61)	The network is reset by NMT command <b>Reset Communication (all nodes)</b> .
NTW_WAIT ( = 0x62)	The CANopen Manager waits for a configurable time so that the modules can execute the <b>Reset Communication</b> command. It uses this time to reset all structures and object entries that are configured during Auto Configuration. After this timeout has expired, the CANopen Manager begins to scan the network. This timeout and the SDO timeout should not be selected too short, in order to prevent a transmit queue overrun when scanning the network.
BOOT_AUTO ( = 0x65)	The CANopen Manager scans the network and creates its own network configuration.
GETPI_INFO ( = 0x66)	The CANopen Manager expects the allocation of the process images to be read out now via the function <b>CMM_GetPIdescr()</b> . When this has happened, the display of the network state is updated.

The following describes how individual objects are configured in the CANopen Manager during the auto configuration mode:

Object entry	Description
NMTStartup [1F80]	The CANopen Manager may not start the modules or itself, so that the following value is configured in the <b>NMTStartup</b> object: NMTStartup = 0x1F
Consumer Heartbeat list [1016]	If a module supports the Heartbeat mechanism, its <b>Producer heartbeat time</b> is configured with a defined value. The module is entered in the <b>Consumer heartbeat time</b> of the CANopen Manager.

## States of the CANopen Manager

---

SlaveAssignment [1F81]	Every detected module is configured as an optional slave. The boot bit is set: Byte 0 of its <b>SlaveAssignment = 0x05</b> If a module supports the node guarding mechanism, the lifetime factor and the guard time of its slave assignment are configured. If a module supports the heartbeat mechanism, the lifetime and guard time of its slave assignment are initialized with 0.
Identity objects [1F84]..[1F88]	The CANopen Manager reads the identity objects of a module and enters them in the relevant object entries of its configuration list. If a module does not support an optional identity sub-index not, 0 (don't care) is entered.
Expected ConfigurationDate, Expected ConfigurationTime [1F26], [1F27]	These entries are initialized with 0, i.e. the module is reconfigured with each boot slave process. During auto configuration, only the default configuration of the module was loaded. This default configuration serves the user as a basis for his/her project-specific configuration.
Concise DCF [1F22]	The concise DCF of a module comprises the values for the error control service (see <b>Consumer heartbeat time</b> or <b>SlaveAssignment</b> ). If the profile type of the module is 401 and it supports the object entry [6423sub00] ( <b>Analog Input Global Interrupt Enable</b> ), this is set to TRUE and this entry is adopted in its concise DCF.
PDOs [1400]..[1BFF]	The CANopen Manager creates for each RPDO scanned on a remote device a corresponding TPDO on itself, and for each valid TPDO scanned on the remote device a corresponding RPDO. The PDO lists of the CANopen Manager are written consecutively with the valid PDO entries. A maximum number of 8 RPDOs or 8 TPDOs are scanned per module. Reading of a PDO table by the CANopen Manager is also aborted if a definable number of PDOs defined as invalid are detected in a module.

### Restrictions:

All object entries of the CANopen Manager that concern the configuration of the network initialization cannot be reconfigured during the auto configuration phase:

- NMTStartup Object [1F80]
- SlaveAssignment Object [1F81]
- Consumer Heartbeat list: Object [1016]
- Identity objects: Objects [1F84]..[1F88]
- Concise DCF Object [1F22]
- ConfigureSlave Object [1F25]
- ExpectedConfigurationDate/Time: Objects [1F26], [1F27]
- BootTime: Object [1F89]  
The boot time can also be reconfigured in the sub-state *BOOT\_END\_MISSING\_MAND* of the CANopen Manager, in order to end or prolong this sub-state.

## States of the CANopen Manager

---

During the auto configuration phase, all object dictionary entries of the individual modules or of the CANopen Manager can be read via SDO access functions. However, the object entries configured during auto configuration are only consistent when the network has been completely scanned.

The states of individual slave modules cannot be altered by NMT functions or by the **RequestNMT** object. The only exceptions are **Reset Communication** or **Reset Node**, which can alter the state of the complete network including the CANopen Manager itself.



**The data collected in the auto configuration phase and the correspondingly modified object entries are not automatically saved in a non-volatile memory after completion of the auto configuration process. They are therefore lost the next time the application program is run or after the function CMM\_InitBoard() has been called.**

### 8.1.5 Network: Scanned

This state is reached after the network is scanned. The boot-up procedure is completed except for the services *Start CANopen Manager* and *Start Network*.

Description of the network state:

Network state	Value	Description
Scanned:	0x8x	The CANopen Manager has scanned the network:
<b>Pre-operational</b>		The lower value 4 bits breaks down the general state further.
	Bit 0 = 1	The state of the network does not correspond to the configuration. Sources of errors: <ul style="list-style-type: none"><li>Optional modules are missing</li><li>Optional modules do not correspond to the configuration: e.g. identity error</li><li>There are unexpected modules in the network</li></ul>
	Bit 1 = 1	The state of the network does not correspond to the configuration. Sources of errors: <ul style="list-style-type: none"><li>Mandatory modules are missing</li><li>Mandatory modules do not correspond to the configuration: e.g.: identity error</li></ul>
	Bit 2 = 1	At least one slave module is "operational"
	Bit 3 = 1	The local slave of the CANopen Manager is "operational"

NMT functions or the **RequestNMT** object, error control events and successful boot slave processes influence the network state.

The state *Network: Scanned* comprises the following states of the boot-up procedure:

Sub-state	Description
Start Manager	In this state the CANopen Manager checks whether it may start itself or not.
Start Network	The CANopen Manager checks whether it may start the network. If so, all modules that are correctly initialized are started.

The behavior of the CANopen Manager is determined by the bits of the **NMTStartup** object.

### Condition:

The CANopen Manager and the network may only be started if all mandatory modules are correctly initialized.

### Restrictions:

All object entries of the CANopen Manager which concern the configuration of the network initialization cannot be configured during the "Network: Scanned" state:

- NMTStartup Object [1080]
- SlaveAssignment Object [1F81]
- Consumer heartbeat time Object [1016]
- Identity objects Objects [1F84]..[1F88]
- Concise DCF Object [1F22]
- ConfigureSlave Object [1F25]
- ExpectedConfigurationDate/Time:  
Objects [1F26], [1F27]

In the *Network: Scanned* state, read access to the object dictionary of the CANopen Manager and to the object directories of the slave modules can be accessed via the SDO access functions.



**The CANopen Manager supports all NMT commands in the Network: Scanned state, i.e. also the commands which concern the complete network.**

### 8.1.6 Network: Operational

The main state of the CANopen network is **Operational**. The network was set to **operational** by the client application via an NMT function or via the **RequestNMT** object.

- The error control service is active
- Detected slave modules are initialized by the boot slave process, in so far as this is allowed by the configuration of its slave assignment
- Emergency and boot-up messages are received

Both error control events and successful boot slave processes influence the network state displayed.

#### Description of the network state:

Network state	Value	Description
Operational	0xAx	The network was set to <b>Operational</b>
Bit 0 = 1:		The state of the network does not correspond to the configuration. Sources of errors: <ul style="list-style-type: none"><li>▪ Optional modules are missing</li><li>▪ Optional modules do not correspond to the configuration, e.g. due to identity errors</li><li>▪ There are unexpected modules in the network</li></ul>
Bit 1 = 1:		The state of the network does not correspond to the configuration. Sources of errors: <ul style="list-style-type: none"><li>▪ Mandatory modules are missing</li><li>▪ Mandatory modules do not correspond to the configuration, e.g. due to identity errors</li></ul> <p>This state occurs when the network was <b>Operational</b>, an error control event of a mandatory module occurred and the configuration of the <b>NMTStartup</b> object allows this state.</p>
Bit 2 = 1:		At least one slave module is <b>Operational</b>
Bit 3 = 1:		The local slave of the CANopen Manager is <b>Operational</b>



The state of individual modules can be altered by NMT functions and by the **RequestNMT** object in the state **Network: Operational**. The general main state of the complete network is not altered by this.

The network can therefore also contain modules whose state is **Pre-operational** or **Stopped**. The diagnostics interface offers a more detailed state description of the individual modules.





A slave module is set to the state requested by the slave assignment or the client application (NMT functions) after a successful boot slave process. The state alteration requested here has priority over the main state of the network.

This information also applies to the states **Network: Stopped** and **Network: Pre-Operational**.

### 8.1.7 Network: Stopped

The main state of the network is **Stopped**. The network management is carried out with restrictions, i.e. all services of the CANopen Manager which work with SDO services cannot be carried out (e.g. boot slave process). The Error Control service is always active. Emergency and boot-up messages are received.

When the boot-up message of a module has been received, this is set to the **Stopped** state, in so far as no other state (e.g. **Pre-operational**) was explicitly requested.

As long as the state of the CANopen Manager is *Network: Stopped*, can neither its object dictionary nor that of the other modules can be accessed.

#### Description of the network state:

Network state	Value	Description
Stopped	0xCx	The network was set to <b>Stopped</b>
	Bit 0 = 1:	The state of the network does not correspond to the configuration. Sources of errors: <ul style="list-style-type: none"> <li>▪ Optional modules are missing</li> <li>▪ Optional modules do not correspond to the configuration, e.g. due to identity errors</li> <li>▪ There are unexpected modules in the network</li> </ul>
	Bit 1 = 1:	The state of the network does not correspond to the configuration. Sources of errors: <ul style="list-style-type: none"> <li>▪ Mandatory modules are missing</li> <li>▪ Mandatory modules do not correspond to the configuration, e.g. due to identity errors</li> </ul>
	Bit 2 = 1:	At least one slave module is <b>Operational</b>
	Bit 3 = 1:	The local slave of the CANopen Manager is <b>Operational</b>



As long as the CANopen Manager is set to **Stopped**, no boot slave processes are carried out. Modules that have registered with their boot-up message are registered and added to the boot list if their **SlaveAssignment** provides for automatic booting.

The CANopen Manager cannot communicate via SDOs in the Network: **Stopped** state. Therefore the search for missing modules is omitted.

If the CANopen Manager was set to another state by a corresponding NMT command, the search for missing modules is continued and the modules waiting to be booted are booted. In this case the boot slave process first sets a module to the **Pre-operational** state.

### 8.1.8 Network: Pre-operational

The main state of the network is **Pre-operational**. The network management is active.

#### Description of the network state:


Network state	Value	Description
Pre-operational	0xEx	The network was set to <b>Pre-operational</b>
	Bit 0 = 1:	The state of the network does not correspond to the configuration. Sources of errors: <ul style="list-style-type: none"><li>Optional modules are missing</li><li>Optional modules do not correspond to the configuration, e.g. due to identity errors</li><li>There are unexpected modules in the network</li></ul>
	Bit 1 = 1:	The state of the network does not correspond to the configuration. Sources of errors: <ul style="list-style-type: none"><li>Mandatory modules are missing</li><li>Mandatory modules do not correspond to the configuration, e.g. due to identity errors</li></ul>
	Bit 2 = 1:	At least one slave module is <b>Operational</b>
	Bit 3 = 1:	The local slave of the CANopen Manager is <b>Operational</b>

### 8.1.9 Slave mode: Pre-operational

In this state the CANopen Manager is configured as a slave. Communication is possible via all communication objects except PDOs.

The object dictionary of the CANopen Manager can be configured by SSSDO via the CAN bus.

The diagnostics interface indicates the state of the modules that are entered in the consumer Heartbeat list [1016] of the CANopen Manager and whose consumer Heartbeat time is not equal to 0.


 **The local SDO access functions allow write only access to the entries TaskTime object ([5F80], mean run time of the main task of the CANopen Manager) and NMTStartup object. The object dictionary of the CANopen Manager is otherwise read only. In slave mode the object directories of other modules cannot be accessed.**

### 8.1.10 Slave mode: operational

In this state the CANopen Manager is configured as a slave. Communication is allowed via all communication objects.

The object dictionary of the CANopen Manager can be configured by SSDO via the CAN bus.

The diagnostics interface indicates the state of the modules that are entered in the **Consumer heartbeat time** object [1016] of the CANopen Manager and whose consumer heartbeat time is not equal to 0.

 **The local SDO access functions allow write only access to the TaskTime object [5F80] and the NMTStartup object. The object dictionary of the CANopen Manager is otherwise read only.**

### 8.1.11 Slave Mode: Stopped

In this state the CANopen Manager is configured as a slave. The CANopen Manager cannot communicate with SDOs or with PDOs here. The local SDO access functions are not available either.

The diagnostics interface indicates the state of the modules that are entered in the **Consumer heartbeat time** object [1016] of the CANopen Manager and whose consumer heartbeat time is not equal to 0.

### 8.1.12 Fatal Error

The CANopen Manager goes into this state if it has detected such a serious error that communication with the network was interrupted.

If the CANopen Manager is configured as a master, it sets the network to the **Stopped** state.

### 8.2 Description of the State Transitions

The following table describes the cause of the individual state transitions. A distinction is made between event-controlled and function-controlled transitions.

Transition	Description
(0)	<ul style="list-style-type: none"><li>▪ The state transition of the CANopen Manager occurs automatically after Power-On</li></ul>
(1)	<ul style="list-style-type: none"><li>▪ Initialization of the CANopen Manager was successful.</li><li>▪ Initialization of the master mode was requested by the application</li><li>▪ The CANopen Manager has registered on the network with its boot-up message.</li><li>▪ The CANopen Manager is <b>Pre-operational</b></li><li>▪ The CANopen Manager is configured as a master:<ul style="list-style-type: none"><li>▪ <b>NMTStartup</b> object [1F80]: bit 0 = 1</li></ul></li></ul>
(2)	<ul style="list-style-type: none"><li>▪ Initialization of the CANopen Manager was successful.</li><li>▪ The CANopen Manager has registered on the network with its boot-up message.</li><li>▪ The CANopen Manager is configured as a slave:<ul style="list-style-type: none"><li>▪ <b>NMTStartup</b> object [1F80]: bit 0 = 0</li><li>▪ <b>NMTStartup</b> object [1F80]: bit 2 = 1:<ul style="list-style-type: none"><li>⇒ The CANopen Manager is <b>Pre-operational</b></li></ul></li><li>▪ The CANopen Manager is in the state <i>Slave mode: pre-operational</i></li><li>▪ <b>NMTStartup</b> object [1F80]: bit 2 = 0:<ul style="list-style-type: none"><li>⇒ The CANopen Manager is <b>Operational</b></li></ul></li><li>▪ The CANopen Manager is in the state <i>Slave mode: operational</i></li></ul></li></ul>
(3)	<ul style="list-style-type: none"><li>▪ The value of the <b>NMTStartup</b> object [1F80] was altered by SSDO: the master functionality was deactivated: bit 0 : 1 → 0</li></ul>
(4)	<ul style="list-style-type: none"><li>▪ The value of the <b>NMTStartup</b> object [1F80] was altered by SSDO: the master functionality was deactivated: bit 0 : 1 → 0</li></ul>

Overview of the various transition groups:

- |             |  |
|-------------|--|
| (5) – (19)  | The CANopen Manager is configured as a master. It initializes or controls the network in accordance with CANopen CiA 302.  |
| (20) – (25) | The CANopen Manager is configured as a slave.  |
| (26)        | <p>Initialization of the CANopen Manager was requested:</p> <ul style="list-style-type: none"> <li>▪ by means of <b>CMM_InitFirmware()</b> by the client application</li> <li>▪ by means of NMT command by the network master if the CANopen Manager is configured as a slave</li> <li>▪ by a <b>RequestNMT</b> command if the CANopen Manager is configured as a master</li> <li>▪ by an error control event of a mandatory module:<br/>the configuration of the <b>NMTStartup</b> object requested a reset of the complete network including the CANopen Manager.</li> </ul> <p>The CANopen Manager had already begun with the initialization of the network</p> |
| (27)        | At least one fatal error was detected by the CANopen Manager:  |
| (28)        | Initialization of the CANopen Manager was requested by the application after the CANopen Manager had signaled the occurrence of a fatal error.   |

The CANopen Manager is configured as a master and manages the network in accordance with CiA 302 (transition groups (5) – (19) )

- |      |  |
|------|--|
| (5)  | <ul style="list-style-type: none"> <li>▪ The application requests initialization of the network in accordance with CiA 302</li> <li>▪ The boot time is started</li> </ul>  |
| (6)  | <ul style="list-style-type: none"> <li>▪ The application requests initialization of the network in auto configuration mode</li> </ul>  |
| (7)  | <ul style="list-style-type: none"> <li>▪ Scanning of the network is completed.</li> <li>▪ The CANopen Manager has scanned and initialized the network.</li> <li>▪ The CANopen Manager itself is not yet started.</li> <li>▪ All mandatory modules are initialized or the boot time has expired</li> </ul>  |
| (8)  | <ul style="list-style-type: none"> <li>▪ Scanning of the network is completed.</li> <li>▪ The CANopen Manager has scanned and initialized the network.</li> <li>▪ The CANopen Manager is already started by an <b>RequestNMT</b> command or by the client application via NMT function.</li> <li>▪ All mandatory modules are initialized .</li> </ul>  |
| (9)  | <ul style="list-style-type: none"> <li>▪ Scanning of the network in auto configuration mode is completed.</li> <li>▪ The CANopen Manager has scanned the complete network scanned and created a configuration for itself</li> <li>▪ The application knows the allocation of the process image</li> </ul>   |
| (10) | <p>The CANopen Manager is <b>Operational</b>, as</p> <ul style="list-style-type: none"> <li>▪ the application set the CANopen Manager to <b>Operational</b> by command</li> <li>▪ the configuration of the <b>NMTStartup</b> object allowed the CANopen Manager to start itself at the end of network initialization</li> <li>▪ the CANopen Manager was set to <b>Operational</b> via <b>RequestNMT</b></li> </ul> <p>Condition: all mandatory modules must be correctly initialized</p> |

## States of the CANopen Manager

---

With the transitions 11/12/13, the boot-up procedure of the network initialization is completed.

- (11) The network including the CANopen Manager was set to **Operational** because
- the client application set the network to **Operational** via NMT function
  - the configuration of the **NMTStartup** object allowed the CANopen Manager to start the network after it was **Operational** itself
  - the network was set to **Operational** via **RequestNMT**
- Condition: all mandatory modules must be correctly initialized
- 

- (12) The network including the CANopen Manager was set to **Stopped** because
- the client application set the network to **Stopped** via NMT function
  - the network was set to **Stopped** via **RequestNMT**
- 

- (13) The network including the CANopen Manager was set to **Pre-operational** because
- the client application set the network to **Pre-operational** via NMT function
  - the network was set to **Pre-operational** via **RequestNMT**

The transitions (14) – (19) refer to state transitions of the complete network including the CANopen Manager.

- (14) The network including the CANopen Manager was set to **Operational** because
- the client application set the network to **Operational** via NMT function
  - the network was set to **Operational** via **RequestNMT**
- Condition: all mandatory modules must be correctly initialized
- 

- (15) The network including the CANopen Manager was set to **Stopped** because
- the client application set the network to **Stopped** via NMT function
  - the network was set to **Stopped** via **RequestNMT**
  - an error control event of a mandatory module occurred and the configuration of the **NMTStartup** object stipulates this reaction
- 

- (16) The network including the CANopen Manager was set to **Pre-operational** because
- the application set the network to **Pre-operational** via command
  - the network was set to **Pre-operational** via **RequestNMT**
- 

- (17) The network including the CANopen Manager was set to **Operational** because
- the client application set the network to **Operational** via NMT function
  - the network was set to **Operational** via **RequestNMT**
- Condition: all mandatory modules must be correctly initialized
- 

- (18) The network including the CANopen Manager was set to **Pre-operational** because
- the client application set the network to **Pre-operational** via NMT function
  - the network was set to **Pre-operational** via **RequestNMT**
- 

- (19) The network including the CANopen Manager was set to **Stopped** because
- the client application set the network to **Stopped** via NMT function
  - the network was set to **Stopped** via **RequestNMT**
-

The transitions (20) – (25) refer exclusively to the *Slave Mode*

- |      |  |
|------|--|
| (20) | ▪ The CANopen Manager was set by NMT command to <b>Pre-operational</b> |
| (21) | ▪ The CANopen Manager was set by NMT command to <b>Operational</b>     |
| (22) | ▪ The CANopen Manager was set by NMT command to <b>Operational</b>     |
| (23) | ▪ The CANopen Manager was set by NMT command to <b>Stopped</b>         |
| (24) | ▪ The CANopen Manager was set by NMT command to <b>Pre-operational</b> |
| (25) | ▪ The CANopen Manager was set by NMT command to <b>Stopped</b>         |

## 9 CANopen Manager API – Functionality Summary

The functionality and the performance of the CANopen Manager API is largely dependent on the CANopen Manager firmware, and thus on the available memory and the micro controller on the CAN board.

The functional characteristics of the CANopen Manager firmware running on a iPC-I XC16 PCI (order no. 1.01.0047) CAN board are listed below:

<b>Description</b>	<b>Value</b>	<b>Unit</b>
PI Input Buffer	2048	Byte
PI Output Buffer	2048	Byte
Maximum number of RPDOs	254	
Maximum number of TPDOs	254	
Concise DCF memory	2048	Byte
Maximum number of dynamically created object dictionary entries	100	
Maximum size of a dynamically created object dictionary entry	8	Byte
Maximum reset time of a slave connected to the CANopen Manager	3	sec
Auto configuration mode: Maximum number of TPDOs and RPDOs read out per slave	8 each	
Auto Configuration mode: Life time factor until detection of a failed slave	4	
NMT: Timeout after command Reset Communication (with boot-up procedure)	2000	ms
Scan mechanism: Cycle time, how often a missing slave is searched for by SDO	1000	ms
SYNC producer: Minimum SYNC cycle time	25	ms
Length of the error list (object [1003])	2	Entries
Size of the emergency queues in the CANopen Manager for each individual slave (object [5F18])	10	Entries
TPDO: maximum number of TPDOs in which the same object can be mapped	2	
Maximum number of objects that can be mapped in a PDO	8	
Number of server SDOs	1	



## CANopen Manager API – Functionality Summary

Description	Value	Unit
Number of client SDOs	3	
CSDO timeout during network initialization	30	ms
CSDO timeout after network initialization	100	ms
CSDO timeout with command SAVE	15000	ms
CSDO timeout with command LOAD	3000	ms

The following table summarizes the functionality supported by the CANopen Manager:

Description	Supported
Non-expedited SDO transfer	yes
SDO block transfer	no
Manager is CANopen heartbeat producer	yes
Manager can be monitored via node guarding	no
Support of synchronous PDOs	yes
Manager is SYNC producer	yes
Manager can transmit emergency objects	yes
Multiplexed PDOs	no
Support of PDO inhibit time	yes
Support of PDO dummy mapping	yes
Dynamic PDO mapping	yes
Support of event-timed PDOs	yes
Support of SDO manager functionality	no
Support of Store/Restore to store values in a non-volatile memory (Flash or EEPROM)	no
Support of Store/Restore to store values in a volatile memory (RAM)	yes
Support of Layer Setting Service (LSS)	no

# 10 CANopen Manager API DLL

The CANopen Manager API provides functions for controlling the CANopen Manager and exchanging data between PC-application and CANopen Manager and the CANopen network.

The CANopen Manager API DLL loads the firmware into the memory of the board, creates the communication structures to the firmware and provides the interface to the PC-application.

The CANopen Manager firmware is configured and parameterized in dialog form via the command interface and the CSDO interface.

As soon as the CANopen Manager is configured, the CANopen network can be booted with a function call. The configuration and start-up of the CANopen slave and error monitoring of the slaves are then carried out independently by the firmware without further intervention by the client application.

In addition, the object directories of external CANopen slaves can be accessed via the SDO functions.

The client application is provided with the status information of the CANopen Manager and of the external CANopen slaves via the diagnostics buffer.

The process data are accessed via the PI buffer. Process data received by the CANopen network via RPDO are supplied to the API-DLL via the PI input. Via the output process image data are transmitted from the API-DLL to the CANopen Manager via the PI output and then sent to the CANopen network by TPDO.

## 10.1 Function Categories

The functions of the CANopen Manager API are pure C `__stdcall` functions and can be divided into the following categories.

- Basic functions
- General functions
- Functions for network management
- Object dictionary and SDO-related functions
- Process image-related functions

### 10.1.1 Basic functions

These functions are used for initialization and parameterization of the API, for selection of the CAN board and for communication with the firmware on the CAN board. The functions are described in detail in section 11.1.

- ***Selection of the CAN board***
  - CMM\_InitBoard
  - CMM\_ReleaseBoard
  - CMM\_GetBoardInfo
  
- ***Initialization and parameterization of the API***
  - CMM\_InitFirmware
  - CMM\_DefineCallbacks
  - CMM\_ResetDLL
  - CMM\_SetCommTimeout
  - CMM\_SetInspecInterval
  - CMM\_DefineMsgProcImg
  - CMM\_DefineMsgMaster
  - CMM\_DefineMsgSlaves
  - CMM\_DefineMsgEvent
  - CMM\_DefineMsgEmergency

### 10.1.2 General functions

The general functions provide state information on the state of the CANopen Manager and of the external CANopen slaves. The functions are described in detail in section 11.2.

- ***Status information***
  - CMM\_GetMasterStat
  - CMM\_GetSlavesStat
  
- ***Other***
  - CMM\_GetEvent
  - CMM\_GetEmergencyObj
  - CMM\_SendEmergencyObj
  - CMM\_HandShake

### 10.1.3 Functions for network management

These are functions for controlling the CANopen Manager and the external CANopen slaves. The functions are described in detail in section 11.2.6.

- ***Boot-up functions***
  - CMM\_StartBootupProc
  - CMM\_StartAutoConfig
  
- ***General network management functions***
  - CMM\_StartNode
  - CMM\_StopNode
  - CMM\_EnterPreOp
  - CMM\_ResetComm
  - CMM\_ResetNode

### 10.1.4 Object dictionary and SDO-related functions

These functions allow to dynamically generate object dictionary entries and for data exchange by means of SDO. A detailed description can be found in section 11.4.

- ***Generation of an object dictionary entry***
  - CMM\_CreateODentry
  
- ***SDO access functions***
  - CMM\_ReadSDO
  - CMM\_WriteSDO
  - CMM\_ReadLocSDO
  - CMM\_WriteLocSDO
  
- ***Import Consice DCF***
  - CMM\_ImportCDC

### 10.1.5 Process image-related functions

These functions enable access to the process image. A discussion of the process image functions is available in section 11.5.

- ***Access to the process image***
  - CMM\_FormPILUT
  - CMM\_GetPIdescr
  - CMM\_GetPI
  - CMM\_GetPIentry
  - CMM\_GetPIIvalue
  - CMM\_PutPIO
  - CMM\_PutPIOentry
  - CMM\_PutPIOvalue
  - CMM\_GetPIIRPDOno
  - CMM\_TriggerPIOoffset

# 11 Individual Functions of the API-DLL

This section contains the complete function reference of the CANopen Manager API. The function prototypes are found in the header file XatCMM.h.

## 11.1 Basic Functions

### 11.1.1 CMM\_InitBoard

**Description:** With `CMM_InitBoard()` an IXXAT CAN board is assigned for use by the CANopen Manager API. For this the CAN board is reset, the CANopen Manager firmware is loaded onto the board and started, the communication queues and buffers are set up and the threads of the CANopen Manager API DLL are created.

If the function has executed successfully, a board handle is returned, which unmistakably identifies the board. This board handle is the first parameter transmitted with every function of the CANopen Manager API.

This function must always be called exactly once by the client program for each board to be used.

**Prototype:**

```

tCMM_ERROR CMM_InitBoard( tCMM_HANDLE*   phBoardhdl,
                           GUID*          pBoardtype,
                           GUID*          pBoardID );
    
```

**Parameters:**

Parameter	Dir.	Explanation
<code>phBoardhdl</code>	(out)	Identifies the board with all following function calls.
<code>pBoardtype</code>	(in/out)	Type of the CAN board. The following values are valid for the CAN board selection (file <code>vciguide.h</code> ): <b>GUID_IPCIXC16PCI_DEVICE</b> <b>GUID_IPCIXC16PCIE_DEVICE</b> There are two special values: <b>CMM_DEFAULTBOARD</b> means that the so-called standard CAN board defined in the IXXAT VCI2 Control Panel Applet (and marked blue there) is to be used. With VCI3, simply the only one installed CAN board will be used. This is the typical application with exactly one CAN board in the computer. <b>CMM_BOARDDIALOG</b> means that a board selection dialog is to be displayed, from which the user can then select the CAN board to be used himself/herself. Value is always returned with <b>CMM_DEFAULTBOARD</b> and <b>CMM_BOARDDIALOG</b> and can be stored for example in the INI-file of the client application.

## Individual Functions of the API-DLL

---

pBoardID	(in/out)	Unique identifier of a CAN board. Is used together with <b>pBoardtype</b> in order to unmistakably identify a board locally. If only one board of the corresponding type is present, the value <b>CMM_1stBOARD</b> is to be given. More than one installed CAN board of the same type can be detected and differentiated by the client application giving consecutive <b>pBoardID</b> values in each case for the same <b>pBoardtype</b> (beginning with 0). In this way the n-th board of this type is taken. For this purpose, there are appropriate tokens <b>CMM_1stBOARD</b> , <b>CMM_2ndBOARD</b> , <b>CMM_3rdBOARD</b> etc already #defined in the main header <b>XatCMM.h</b> . Value is always returned and can be stored for example in the INI-file of the client application.
----------	----------	--

---

### Return values:

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_BOARD_ALREADY_USED	Board is allocated
CMMERR_ALL_BOARDS_USED	No free board available
CMMERR_CANNOT_SEARCH_BOARD	IXXAT hardware selection dialog was ended with abort
CMMERR_BOARD_NOT_FOUND	Board with specified board type (pBoardtype) and key (pRegkey) not found
CMMERR_BOARD_NOT_SUPP	Specified board type not compatible with CANopen Manager
CMMERR_WRONG_FW	Incorrect firmware version or communication with firmware failed
CMMERR_USED_FROM_OTHER_PROCESS	Board is being used by another CAN application
CMMERR_PC_MC_COMM_ERR	Communication between PC and CAN board failed
CMMERR_BOARD_DLD_ERR	Error during firmware download
CMMERR_NO_SUCH_CANLINE	CAN line is not supported
CMMERR_CANLINE_USED	CAN line is already being used
CMMERR_VCI_INST_ERR	IXXAT VCI driver missing
CMMERR_BOARD_ERR	Unknown board type or board type cannot be localized
CMMERR_CCI_INST_ERR	Internal CCI instance error
CMMERR_SDO_INST_ERR	Internal SDO manager installation error



### 11.1.2 CMM\_ReleaseBoard

**Description:** With `CMM_ReleaseBoard()` an IXXAT CAN board allocated by the CANopen Manager API is released and reset. This function must always be called by the client program exactly once for each board used.

**Prototype:** `tCMM_ERROR CMM_ReleaseBoard( tCMM_HANDLE hBoard );`

**Parameter:**

Parameter	Dir.	Explanation
Hboard	(in)	Handle of the CAN board

**Return values:**

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle

### 11.1.3 CMM\_GetBoardInfo

**Description:** With `CMM_GetBoardInfo()` information on the hardware properties of the CAN board used and the version numbers of the software components are requested.

**Prototype:** `tCMM_ERROR CMM_GetBoardInfo(  
tCMM_HANDLE hBoard,  
tCMM_BOARDINFO* pBoardInfo );`

**Parameters:**

Parameter	Dir.	Explanation
<code>hBoard</code>	(in)	Handle of the CAN board
<code>pBoardInfo</code>	(in/out)	Pointer to information structure
<code>tCMM_BOARDINFO</code>		Alignment: 1 byte
Structural element	Type	Meaning
<code>HWversion</code>	WORD	Version number of the CAN board e.g. 0x0101-> V 1.01
<code>FWversion</code>	WORD	Version number of the CANopen Manager firmware e.g. 0x0132-> V 1.32
<code>SWversion</code>	WORD	Version number of the CANopen Manager API e.g. 0x0400-> V 4.00
<code>Swbuild</code>	WORD	CANopen Manager API build version
<code>BoardSeg</code>	DWORD	I/O address of the board used
<code>BoardIRQ</code>	WORD	Interrupt request line IRQ used by the board
<code>BoardCANS</code>	WORD	Number of supported CAN lines of the board
<code>BoardSerialNo[16]</code>	char[]	Serial number of the CAN board as string
<code>BoardType[40]</code>	char[]	Description of the board type as string
<code>FWtarget</code>	eCMM_FW TARGET	VCI2 resp VCI3 firmware is running from RAM or FLASH.

**Return values:**

Return value	Description
<code>CMMERR_OK</code>	Success
<code>CMMERR_INVALID_HANDLE</code>	Invalid board handle
<code>CMMERR_NOT_SENT</code>	Command interface allocated, command not transmitted, another attempt necessary
<code>CMMERR_COMMTIMEOUT</code>	Timeout of the command interface
<code>CMMERR_INVALID_PARAM</code>	Invalid function parameter
<code>CMMERR_NO_OBJECTS</code>	No objects in queue
<code>CMMERR_CCI_INST_ERR</code>	CCI installation error (internal)

### 11.1.4 CMM\_InitFirmware

**Description:** With `CMM_InitFirmware()` the CANopen Manager firmware is initialized. All data structures are reset and the CAN controller is initialized. This is the "initialization" state of the CANopen Manager.

**Prototype:**

```

tCMM_ERROR CMM_InitFirmware(
    tCMM_HANDLE hBoard,
    BYTE InitMode,
    BYTE Baudrate,
    BYTE NodeNo,
    WORD HsInterval,
    WORD HSReaction );
```

**Parameters:**

Parameter	Dir.	Explanation																
HBoard	(in)	Handle of the CAN board																
InitMode	(in)	<p>COP_k_RESETNODE: Reset of the CANopen Manager</p> <p>COP_k_RESETCOM: Reset only the communication parameters of the CANopen Manager</p> <p>The function must be called with the first call with the value COP_k_RESETNODE!</p>																
Baudrate	(in)	<p>Bit rate of the CAN. The following values are permitted:</p> <table border="0"> <tr> <td>CMM_BAUDRATE_10</td> <td>10 kBit/s</td> </tr> <tr> <td>CMM_BAUDRATE_20</td> <td>20 kBit/s</td> </tr> <tr> <td>CMM_BAUDRATE_50</td> <td>50 kBit/s</td> </tr> <tr> <td>CMM_BAUDRATE_100</td> <td>100 kBit/s</td> </tr> <tr> <td>CMM_BAUDRATE_125</td> <td>125 kBit/s</td> </tr> <tr> <td>CMM_BAUDRATE_250</td> <td>250 kBit/s</td> </tr> <tr> <td>CMM_BAUDRATE_500</td> <td>500 kBit/s</td> </tr> <tr> <td>CMM_BAUDRATE_1000</td> <td>1000 kBit/s</td> </tr> </table>	CMM_BAUDRATE_10	10 kBit/s	CMM_BAUDRATE_20	20 kBit/s	CMM_BAUDRATE_50	50 kBit/s	CMM_BAUDRATE_100	100 kBit/s	CMM_BAUDRATE_125	125 kBit/s	CMM_BAUDRATE_250	250 kBit/s	CMM_BAUDRATE_500	500 kBit/s	CMM_BAUDRATE_1000	1000 kBit/s
CMM_BAUDRATE_10	10 kBit/s																	
CMM_BAUDRATE_20	20 kBit/s																	
CMM_BAUDRATE_50	50 kBit/s																	
CMM_BAUDRATE_100	100 kBit/s																	
CMM_BAUDRATE_125	125 kBit/s																	
CMM_BAUDRATE_250	250 kBit/s																	
CMM_BAUDRATE_500	500 kBit/s																	
CMM_BAUDRATE_1000	1000 kBit/s																	
NodeNo	(in)	Node-ID of the CANopen Manager																
HsInterval	(in)	Maximum Handshake interval in milliseconds, set to 0 if no handshake shall be used, this value + 100 [ms] must be smaller than the communication timeout value set with <code>CMM_SetCommTimeout()</code>																

## Individual Functions of the API-DLL

---

HSReactions	(in)	<p>Reactions of the CANopen Manager firmware when handshake timeout timer elapses</p> <p>Use any of the following NMT related flags (possible only in Master mode):</p> <p>CMM_HS_MASTER_NMT_PREOP_ALL: send NMT message Enter Pre-operational all nodes</p> <p>CMM_HS_MASTER_NMT_STOP_ALL: send NMT message Stop all nodes</p> <p>CMM_HS_MASTER_NMT_RESETCOMM_ALL: send NMT message Reset Communication all nodes</p> <p>CMM_HS_MASTER_NMT_RESETNODE_ALL: send NMT message Reset Node all nodes</p> <p>The following reactions can be freely combined via disjunction:</p> <p>CMM_HS_SEND_EMICY: send an Emergency message</p> <p>CMM_HS_SLAVE_STOP_SENDING_HBT: stops sending of own Heartbeat (possible only in Slave mode)</p>
-------------	------	--

### Return values:

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_NOT_SENT	Command interface allocated, command not transmitted, another attempt necessary
CMMERR_NOT_CONFIGURED	CANopen Manager was not configured
CMMERR_NO_OBJECTS	No data in communication queues
CMMERR_COMMTIMEOUT	Timeout of the command interface
CMMERR_CCI_INST_ERR	CCI installation error
CMMERR_INVALID_PARAM	Invalid function parameter
CMMERR_FW_INIT_FAILED	Failure during initialization of Manager firmware

### 11.1.5 CMM\_DefineCallbacks

**Description:** With `CMM_DefineCallbacks()` the CANopen Manager API is informed of functions from the client application of type `tCMM_CALLBACK` CANopen Manager API which are then called when a specific event occurs.

The prototype of the callback functions is found in section 11.1.6, `tCMM_CALLBACK`. A zero pointer is allowed as a function parameter in order to mark unused callback functions.

**Prototype:**

```
tCMM_ERROR CMM_DefineCallbacks(
    tCMM_HANDLE hBoard,
    tCMM_CALLBACK fpProcImg,
    tCMM_CALLBACK fpMaster,
    tCMM_CALLBACK fpSlaves,
    tCMM_CALLBACK fpNotification,
    tCMM_CALLBACK fpEmergency );
```

**Parameters:**

Parameter	Dir.	Explanation
HBoard	(in)	Handle of the CAN board
fpProcImg	(in)	This function is called when an alteration in the PI input has occurred.
fpMaster	(in)	This function is called when an alteration in the master status has occurred. See also the description of <code>CMM_GetMasterStat</code>
fpSlaves	(in)	This function is called when an alteration in the state of a slave has occurred. See also the description of <code>CMM_GetSlavesStat</code>
fpNotification	(in)	This function is called when an exception has occurred. See also the description of <code>CMM_GetEvent</code>
fpEmergency	(in)	This function is called when an Emergency message was received. See also the description of <code>CMM_GetEmergencyObj</code>

**Return values:**

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_BADCALLBACK_PTR	A callback pointer is invalid

### 11.1.6 tCMM\_CALLBACK

**Description:** tCMM\_CALLBACK is a function prototype for functions within the client application that are registered with the CANopen Manager API with CMM\_DefineCallbacks() and are called to signal events.

**Prototype:**

```
typedef void (CALLBACK* tCMM_CALLBACK)(
    tCMM_HANDLE hBoard,
    DWORD dwSource,
    DWORD dwRes );
```

**Parameters:**

Parameter	Dir.	Explanation
hBoard	(in)	Handle of the CAN board
dwSource	(in)	Identifier of the source of the event
dwRes	(in)	Reserved parameter

**Return values:**

none

### 11.1.7 CMM\_ResetDLL

**Description:** With `CMM_ResetDLL()` the CANopen Manager DLL is reinitialized, in order to be able to re-register the board in an interpreter debugger such as Visual Basic in the event of a program abort (without release of the board).

**All registered CAN boards are deregistered. This function should only be used during program development.**



**Prototype:** `void CMM_ResetDLL();`

**Parameters:**

none

**Return values:**

none

### 11.1.8 CMM\_SetCommTimeout

**Description:** With `CMM_SetCommTimeout()` the timeout is defined, which determines how long acknowledgement of the CANopen Manager firmware is awaited.

With almost every function of the CANopen Manager API a job structure is compiled within the DLL for the master firmware, transmitted to it, and processing or acknowledgement on the part of the master firmware is awaited. If the pre-set timeout is exceeded, the individual function returns with the return value `CMMERR_COMMTIMEOUT`. The default value for the timeout is 5 seconds.

**Prototype:** `tCMM_ERROR CMM_SetCommTimeout(  
tCMM_HANDLE hBoard,  
WORD wTimeout );`

**Parameters:**

Parameter	Dir.	Explanation
<code>hBoard</code>	(in)	Handle of the CAN board
<code>wTimeout</code>	(in)	New value for the timeout in milliseconds. The Value range is $55 \leq w\_timeout \leq 65535$ . Smaller values are rounded up internally. Note. If an handshake interval is set (see <code>CMM_InitFirmware()</code> ) then the new value must be greater than the handshake interval + 100 [ms].

**Return values:**

Return value	Description
<code>CMMERR_OK</code>	Success
<code>CMMERR_INVALID_HANDLE</code>	Invalid board handle
<code>CMMERR_INVALID_PARAM</code>	Timeout value is out of range



### 11.1.9 CMM\_SetInspecInterval

**Description:** CMM\_SetInspecInterval() sets the interval with which the internal status poll threads work.

For each callback function a thread runs internally, which checks cyclically for alterations in the relevant DPRAM buffers. The cycle interval can be selected from 1msec, where the pre-set value is 6ms.

If the poll thread now detects a difference between its last stored value and the current contents of the DPRAM buffer, it first calls the callback function, then it transmits the stored message to the window, after that it transmits the stored message to the application thread and finally saves the read value.

The cycle time applies equally to all poll threads.

**Prototype:** `tCMM_ERROR CMM_SetInspecInterval(  
tCMM_HANDLE hBoard,  
DWORD dwInterval );`

**Parameters:**

Parameter	Dir.	Explanation
Hboard	(in)	Handle of the CAN board
DwInterval	(in)	New value for the thread cycle time in milliseconds. The smallest permitted value is 1 ms.

**Return values:**

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_MEM_ALLOC_ERR	Restart of at least one thread timer failed

### 11.1.10 CMM\_DefineMsgProcImg

**Description:** This function is used to link user-defined messages with the event that PI input has changed.

It is possible for the client application to receive a Windows message, a thread message or both when a change has been detected. The API-DLL will then call the WINAPI functions `PostMessage()` or `PostThreadMessage()`. The messages transmit the CAN board handle as `wParam` and the identifier of the source of the event as `lParam`.

If one of the two messages is not required, `INVALID_HANDLE_VALUE` is to be specified as the corresponding argument.

To read data from the PI input the functions `CMM_GetPI()` or alternatively `CMM_GetPIentry()` are available.

**Prototype:**

```
tCMM_ERROR CMM_DefineMsgProcImg(  
                                tCMM_HANDLE hBoard,  
                                HWND hwnd,  
                                DWORD idThread,  
                                UINT Msg );
```

**Parameters:**

<b>Parameter</b>	<b>Dir.</b>	<b>Explanation</b>
HBoard	(in)	Handle of the CAN board
Hwnd	(in)	Handle of the window to which the Windows messages defined here are to be transmitted
IdThread	(in)	Thread identifier of the target thread
Msg	(in)	Message identifier of the message

**Return values:**

<b>Return value</b>	<b>Description</b>
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle

### 11.1.11 CMM\_DefineMsgMaster

**Description:** This function is used to link user-defined messages with the event that a change in the master state has taken place. The master state is described in section 7.1

It is possible for the client application to receive a Windows message, a thread message or both when a change has been detected. The API-DLL will then call the WINAPI functions `PostMessage()` or `PostThreadMessage()`. The messages transmit the CAN board handle as `wParam` and the identifier of the source of the event as `lParam`.

If one of the two messages is not required, `INVALID_HANDLE_VALUE` is to be specified as the corresponding argument.

To retrieve the state of the CANopen Manager use the API function `CMM_GetMasterStat()`.

**Prototype:**

```

tCMM_ERROR CMM_DefineMsgMaster(
                                tCMM_HANDLE hBoard,
                                HWND        hwnd,
                                DWORD        idThread,
                                UINT        Msg );
    
```

**Parameters:**

Parameter	Dir.	Explanation
HBoard	(in)	Handle of the CAN board
hwnd	(in)	Handle of the window to which the Windows messages defined here are to be transmitted
idThread	(in)	Thread identifier of the target thread
Msg	(in)	Message identifier of the message

**Return values:**

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle

### 11.1.12 CMM\_DefineMsgSlaves

**Description:** This function is used to link user-defined Windows messages with the event that a change in the state of a slave has occurred. The slave state is described in section 7.2

It is possible for the client application to receive a Windows message, a thread message or both when a change has been detected. The API-DLL will then call the WINAPI functions `PostMessage()` or `PostThreadMessage()`. The messages transmit the CAN board handle as `wParam` and the identifier of the source of the event as `lParam`.

If one of the two messages is not required, `INVALID_HANDLE_VALUE` is to be specified as the corresponding argument.

To retrieve the state of a slave device, use the API function `CMM_GetSlavesStat()`.

**Prototype:**

```
tCMM_ERROR CMM_DefineMsgSlaves(  
                                  tCMM_HANDLE hBoard,  
                                  HWND hwnd,  
                                  DWORD idThread,  
                                  UINT Msg );
```

**Parameters:**

<b>Parameter</b>	<b>Dir.</b>	<b>Explanation</b>
<code>hBoard</code>	(in)	Handle of the CAN board
<code>hwnd</code>	(in)	Handle of the window to which the Windows messages defined here are to be transmitted
<code>idThread</code>	(in)	Thread identifier of the target thread
<code>Msg</code>	(in)	Message identifier of the message

**Return values:**

<b>Return value</b>	<b>Description</b>
<code>CMMERR_OK</code>	Success
<code>CMMERR_INVALID_HANDLE</code>	Invalid board handle

### 11.1.13 CMM\_DefineMsgEvent

**Description:** This function is used to correlate user defined Windows messages with an exception. Exception are events like critical state changes of the CANopen Manager. The definitions of the corresponding constants begin with `CMM_NOTI_KIND_` and can be found in the file `XatMMdefs.h`.

It is possible for the client application to receive a Windows message, a thread message or both when a change has been detected. The API-DLL will then call the WINAPI functions `PostMessage()` or `PostThreadMessage()`. The messages transmit the CAN board handle as `wParam` and the identifier of the source of the event as `lParam`.

If one of the two messages is not required, `INVALID_HANDLE_VALUE` is to be specified as the corresponding argument.

An exception may be retrieved with the function `CMM_GetEvent()`.

**Prototype:**

```

tCMM_ERROR CMM_DefineMsgEvent(
                                tCMM_HANDLE hBoard,
                                HWND hwnd,
                                DWORD idThread,
                                UINT Msg );
    
```

**Parameter:**

Parameter	Dir.	Explanation
<code>hBoard</code>	(in)	Handle of the CAN board
<code>hwnd</code>	(in)	Handle of the application window to which the Windows messages defined here will be send
<code>idThread</code>	(in)	Thread ID of the target thread
<code>Msg</code>	(in)	Message ID

**Return values:**

Return value	Description
<code>CMMERR_OK</code>	Success
<code>CMMERR_INVALID_HANDLE</code>	Invalid board handle

### 11.1.14 CMM\_DefineMsgEmergency

**Description:** This function is used to correlate user defined Windows messages with the reception of an Emergency message.

It is possible for the client application to receive a Windows message, a thread message or both when a change has been detected. The API-DLL will then call the WINAPI functions `PostMessage()` or `PostThreadMessage()`. The messages transmit the CAN board handle as `wParam` and the identifier of the source of the event as `lParam`.

If one of the two messages is not required, `INVALID_HANDLE_VALUE` is to be given as the corresponding parameter.

To retrieve the Emergency message from the queue use the function `CMM_GetEmergencyObj()`.

**Prototype:**

```
tCMM_ERROR CMM_DefineMsgEmergencyt(  
                tCMM_HANDLE hBoard,  
                HWND hwnd,  
                DWORD idThread,  
                UINT Msg );
```

**Parameter:**

Parameter	Dir.	Explanation
<code>hBoard</code>	(in)	Handle of the CAN board
<code>hwnd</code>	(in)	Handle of the application window to which the Windows messages defined here will be send
<code>idThread</code>	(in)	Thread ID of the target thread
<code>Msg</code>	(in)	Message ID

**Return values:**

Return value	Description
<code>CMMERR_OK</code>	Success
<code>CMMERR_INVALID_HANDLE</code>	Invalid board handle

## 11.2 General Functions

### 11.2.1 CMM\_GetMasterStat

**Description:** This function supplies the status information of the CANopen Manager. Section 7.1 describes the contents of the function parameters at bit level.

**Prototype:**

```

tCMM_ERROR CMM_GetMasterStat(
                                tCMM_HANDLE hBoard,
                                WORD* pMasterManagerState,
                                WORD* fGlobalEvents,
                                WORD* fConfigBits );
    
```

**Parameters:**

Parameter	Dir.	Explanation
<b>hBoard</b>	(in)	Handle of the CAN board
<b>pMasterManagerState</b>	(out)	Status of the CANopen Manager
<b>fGlobalEvents</b>	(out)	Global events bit field
<b>fConfigBits</b>	(out)	Configuration bits

**Return values:**

Return value	Description
<b>CMMERR_OK</b>	Success
<b>CMMERR_INVALID_HANDLE</b>	Invalid board handle
<b>CMMERR_CCI_INST_ERR</b>	CCI installation error

### 11.2.2 CMM\_GetSlavesStat

**Description:** This function supplies bit fields which show the state of all slaves. Section 7.2 describes the structure of the bit fields and the individual function parameters in detail.

**Prototype:**

```
tCMM_ERROR CMM_GetSlavesStat(  
    tCMM_HANDLE          hBoard,  
    tCMM_SLAVEFLAGS*    fAssigned,  
    tCMM_SLAVEFLAGS*    fConfigured,  
    tCMM_SLAVEFLAGS*    fMismatch,  
    tCMM_SLAVEFLAGS*    fEmergency,  
    tCMM_SLAVEFLAGS*    fOperational,  
    tCMM_SLAVEFLAGS*    fStopped,  
    tCMM_SLAVEFLAGS*    fPreOperational );
```

**Parameters:**

Parameter	Dir.	Explanation
hBoard	(in)	Handle of the CAN board
fAssigned	(out)	Flags of the slaves assigned to the CANopen Manager
fConfigured	(out)	Flags of the fully configured slaves
fMismatch	(out)	Flags of the slaves with incorrect configuration (for example a slave present in the network that was not expected by the CANopen Manager)
fEmergency	(out)	Flags of the slaves from which an emergency message was received
fOperational	(out)	Flags of the slaves in OPERATIONAL state
fStopped	(out)	Flags of the slaves in STOPPED state
fPreOperational	(out)	Flags of the slaves in PRE-OPERATIONAL state

**Return values:**

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_CCI_INST_ERR	CCI installation error



### 11.2.3 CMM\_GetEvent

**Description:** This retrieves one exception event from the event queue. The event may originate from different reasons: A critical error in the state of the CANopen Manager software, which may be triggered by a CAN communication error, or a network event, like the failure of a mandatory slave node.

**Prototype:**

```

tCMM_ERROR CMM_GetEvent( tCMM_HANDLE hBoard,
BYTE* EvtType,
BYTE* EvtData1,
BYTE* EvtData2,
BYTE* EvtData3,
BYTE* EvtData4 );
```

**Parameter:**

Parameter	Dir.	Explanation
hBoard	(in)	Handle of the CAN board
EvtType	(out)	Kind of event. The following values are possible: <b>CMM_NOTI_KIND_FATALERROR</b> Critical error in the CAN communication of the CANopen Manager <b>CMM_NOTI_KIND_GLOBALEVENT</b> Network event <b>CMM_NOTI_KIND_UNEXPECTEDNODESTATE</b> unexpected node state detected <b>CMM_NOTI_KIND_GUARDERROR</b> guard- or heartbeaterror, no response from slave <b>CMM_NOTI_KIND_BOOTUP</b> (unexpected) bootup message received <b>CMM_NOTI_KIND_HANDSHAKETIMEOUT</b> maximum firmware <-> application handshake response time exceeded by application <b>CMM_NOTI_KIND_TRIGGERTPDOQUEUE</b> PI offset given in a call to function <code>CMM_TriggerPIOffset()</code> is invalid <b>CMM_NOTI_KIND_SCANNODEDETECTED</b> Scanning has detected new node <b>CMM_NOTI_KIND_NODECONFIGURED</b> Node is configured
EvtData1	(out)	Additional information of the event
EvtData2	(out)	Additional information of the event
EvtData3	(out)	Reserved
EvtData4	(out)	Reserved

Depending on the contents of the parameter `EvtType` additional information is coded in the four parameters `EvtDataX`. A description of possible parameter values is given in the following tables.

## Individual Functions of the API-DLL

*EvtType == CMM_NOTI_KIND_FATALERROR	
EvtData1	EvtData2
Communication state (coded bit wise), corresponds to HIGHBYTE(wMasterManagerState) of the CANopen Manager State (see also section 7.1.2)	Not used
NMS_k_EV_LPRXOVR Overrun of the low-priority receive queue	
NMS_k_EV_CANOVR CAN controller overrun	
NMS_k_EV_BUSOFF CAN controller im BusOff Zustand	
NMS_k_EV_ESTATSET Error state bit of the CAN controller is set	
NMS_k_EV_ESTATRESET Error state bit of the CAN controller is reset	
NMS_k_EV_LPTXOVR Overrun of the low-priority transmit queue	
NMS_k_EV_HPRXOVR Overrun of the high-priority receive queue	
NMS_k_EV_HPTXOVR Overrun of the high-priority transmit queue	

*EvtType == CMM_NOTI_KIND_GLOBALEVENT	
EvtData1	EvtData2
Network event (bit coded), corresponds to <b>LOWBYTE(wGlobalEvents)</b> of the CANopen Manager state (see also 7.1.3)	Network event (bit coded), corresponds to <b>HIGHBYTE(wGlobalEvents)</b> of the CANopen Manager state (see also section 7.1.3)
<b>FATE</b> Error in the communication with the network	<b>ASE</b> Is set, if the <b>SlaveAssignment-</b> objekt [1F81] for a module contains features not supported by the CANopen Manager
<b>NIDE</b> A module uses the node-ID of the CANopen Manager	<b>PDLEN_ERR</b> The CANopen Manager has received an RPDO with to few data bytes
<b>MSE</b> Error control event of a mandatory slave	<b>CONFIG_ERR</b> A concise DCF is internally incorrect or does not correspond to the object dictionary of the slave device
<b>MNCE</b> Identity error respectively incorrect concise DCF for a mandatory slave	<b>API_ROVR</b> Indication of a queue overrun of the CSDO interface
<b>OIE</b> Identity error of an optional slave	<b>RSCN</b> The NMT state of the entire network has been modified via the <b>RequestNMT</b> objekt
<b>PIE</b> Generation of configuration, process image, and PDOs did not success in auto configuration mode	<b>RSCM</b> The NMT state of an individual module was changed via the <b>RequestNMT</b> object
<b>ACE</b> While scanning the network in auto configuration mode an error event of a already scanned module has been detected, or the boot-up message of a module got detected after the scan of this module	
<b>NMTE</b> Is set as soon as any of the bits in the slave state bitlists has changed	

*EvtType == CMM_NOTI_KIND_UNEXPECTEDNODESTATE	
EvtData1	EvtData2
Node ID of the slave	the unexpected state delivered from the slave

*EvtType == CMM_NOTI_KIND_GUARDERROR	
EvtData1	EvtData2
Node ID of the slave	Not used

## Individual Functions of the API-DLL

*EvtType == CMM_NOTI_KIND_BOOTUP	
EvtData1	EvtData2
Node ID of the slave	Not used

*EvtType == CMM_NOTI_KIND_HANDSHAKETIMEOUT	
EvtData1	EvtData2
executed reaction to application handshake timeout: CMM_HS_SEND_EMCY Emergency message was sent CMM_HS_MASTER_NMT_PREOP_ALL NMT PRE-OPERATIONAL all was sent CMM_HS_MASTER_NMT_STOP_ALL NMT Stop all was sent CMM_HS_MASTER_NMT_RESETCOMM_ALL NMT Reset Communication was sent CMM_HS_MASTER_NMT_RESETNODE_ALL NMT Reset Node all was sent CMM_HS_SLAVE_STOP_SENDING_HBT sending of the own CANopen Heartbeat is stopped	Not used

*EvtType == CMM_NOTI_KIND_TRIGGERTPDOQUEUE	
EvtData1	EvtData2
COP_k_ERR object not found COM_k_NOT_CONFIGURED object found but not mapped	Not used

*EvtType == CMM_NOTI_KIND_SCANNODEDETECTED	
EvtData1	EvtData2
Node ID of the slave	Not used

*EvtType == CMM_NOTI_KIND_NODECONFIGURED	
EvtData1	EvtData2
Node ID of the slave	Not used

### Return values:

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_INVALID_PARAM	Invalid parameter
CMMERR_NO_OBJECTS	No data in the event queue
CMMERR_CCI_INST_ERR	CCI installation error

### 11.2.4 CMM\_GetEmergencyObj

**Description:** This reads out one entry from the receive emergency queue. The CANopen Manager firmware puts all received Emergency objects into this queue. With this function those entries will be read out.

**Prototype:**

```

tCMM_ERROR CMM_GetEmergencyObj(
    tCMM_HANDLE hBoard,
    BYTE* NodeNo,
    WORD* ErrCode,
    BYTE* ErrRegister,
    BYTE* ErrField );
```

**Parameters:**

Parameter	Dir.	Explanation
hBoard	(in)	Handle of the CAN board
NodeNo	(out)	Number of the node that sent the emergency object
ErrCode	(out)	Error code of emergency object
ErrRegister	(out)	Error register value of emergency object
ErrFiled	(out)	Manufacturer specific error field of emergency object (5 bytes)

**Return values:**

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_INVALID_PARAM	Invalid function parameter
CMMERR_NO_OBJECTS	No objects in queues
CMMERR_CCI_INST_ERR	CCI installation error

### 11.2.5 CMM\_SendEmergencyObj

**Description:** This function sends an alarm message of the CANopen Manager. The data field of the corresponding CAN message can be specified with the function parameters, the COB-ID is stipulated by the node number of the CANopen Manager according to **predefined connection set** or by the contents of the object [1014].

**Prototype:** `tCMM_ERROR CMM_SendEmergencyObj(  
tCMM_HANDLE hBoard,  
WORD ErrCode,  
BYTE ErrRegister,  
BYTE* ErrData );`

**Parameters:**

Parameter	Dir.	Explanation
hBoard	(in)	Handle of the CAN board
ErrCode	(in)	Error code of the alarm message (first and second data byte). Contains a pre-defined error code in accordance with CiA 301 "Emergency Error Codes"
ErrRegister	(in)	Third byte of the alarm message: contents of the object [1001]
ErrData	(in)	Byte array of length 5. (fourth to eighth byte of the alarm message). The contents are not stipulated. For this reason, vendor-specific error detection can be incorporated.

**Return values:**

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_NOT_SENT	Command interface allocated, command not transmitted, another attempt necessary
CMMERR_NOT_CONFIGURED	CANopen Manager was not configured
CMMERR_NO_OBJECTS	No data in communication queues
CMMERR_COMMTIMEOUT	Timeout of command interface
CMMERR_CCI_INST_ERR	CCI installation error
CMMERR_GENERAL_ERR	General error
CMMERR_STATE_ERR	Command not allowed in current state of the CANopen Manager
CMMERR_EMICY_INHIBITED	Emergency object not sent because emergency inhibit time still active

### 11.2.6 CMM\_HandShake

**Description:** CMM\_HandShake() executes one handshake with the CANopen Manager firmware.

To enable the handshake timeout timer of the CANopen Manager firmware the handshake interval must be set to an value unequal 0 before using CMM\_InitFirmware(). With the 1<sup>st</sup> call of CMM\_HandShake() the enabled handshake timeout timer starts.

After the handshake timeout timer was started the application must call CMM\_HandShake() within the handshake interval. If the handshake timeout timer elapses the configured reactions will be executed (see CMM\_InitFirmware() argument HsReaction).

**Prototype:** `tCMM_ERROR CMM_HandShake( tCMM_HANDLE hBoard );`

**Parameters:**

Parameter	Dir.	Explanation
hBoard	(in)	Handle of the CAN board

**Return values:**

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_NOT_SENT	Command interface allocated, command not transmitted, another attempt necessary
CMMERR_NOT_CONFIGURED	CANopen Manager was not configured
CMMERR_NO_OBJECTS	No data in communication queues
CMMERR_COMMTIMEOUT	Timeout of the command interface
CMMERR_CCI_INST_ERR	CCI installation error
CMMERR_WRONG_FW	Invalid firmware handshake response

### 11.3 Functions for Network Management

#### 11.3.1 CMM\_StartBootupProc

**Description:** This function starts the boot-up procedure of the CANopen Manager. All slaves are configured using their default SDO. Before the function is called, the CANopen Manager must be configured via the local SDO access functions, for example the process data must be mapped in PDOs and the local object dictionary entries created.  
In the event of a faulty SDO write access to a slave node, the complete boot-up procedure is aborted.  
The network is set to the **PRE-Operational** state.

**Prototype:** `tCMM_ERROR CMM_StartBootupProc(  
tCMM_HANDLE hBoard );`

**Parameter:**

Parameter	Dir.	Explanation
hBoard	(in)	Handle of the CAN board

**Return values:**

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_NOT_SENT	Command interface allocated, command not transmitted, another attempt necessary
CMMERR_NOT_CONFIGURED	CANopen Manager was not configured
CMMERR_NO_OBJECTS	No data in communication queues
CMMERR_COMMTIMEOUT	Timeout of the command interface
CMMERR_CCI_INST_ERR	CCI installation error
CMMERR_NOT_AUTHORIZED	CANopen Manager not local master
CMMERR_STATE_ERR	Command not allowed in current state of the CANopen Manager
CMMERR_SDO_INUSE	An SDO transfer was not completed yet. The function must be called again.



### 11.3.2 CMM\_StartAutoConfig

**Description:** The function `CMM_StartAutoConfig()` starts the auto configuration process of the CANopen Manager. During the boot-up, the slaves detected in the CANopen network are configured with their default SDO. They are reset to their default settings and the NMT monitoring is started. The CANopen Manager automatically maps the data from the PDOs of the slaves in network variables and stores these in the PI input or PI output. The network is set to the **Operational** state and the data exchange begins.

Please ensure that you consult section 5.3 concerning the working method, the conditions and the restrictions of the auto configuration process.

**Prototype:** `tCMM_ERROR CMM_StartAutoConfig(  
tCMM_HANDLE hBoard,  
WORD HeartbeatTime );`

**Parameter:**

Parameter	Dir.	Explanation
hBoard	(in)	Handle of the CAN board
HeartbeatTime	(in)	producer heartbeat time to set for all CANopen slaves in the network

**Return values:**

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_NOT_SENT	Command interface allocated, command not transmitted, another attempt necessary
CMMERR_NOT_CONFIGURED	CANopen Manager was not configured
CMMERR_NO_OBJECTS	No data in communication queues
CMMERR_COMMTIMEOUT	Timeout of the command interface
CMMERR_CCI_INST_ERR	CCI installation error
CMMERR_NOT_AUTHORIZED	CANopen Manager is not local CANopen master
CMMERR_STATE_ERR	Command not allowed in current state of the CANopen Manager
CMMERR_SDO_INUSE	An SDO transfer was not completed yet. The function must be called again.

### 11.3.3 CMM\_StartNode

**Description:** NMT function. This function is used to set a CANopen node (also of the CANopen Manager itself, in so far as its node number is given) or of the complete CANopen network to the **Operational** state using the NMT command **Start Remote Node**.

**Prototype:** `tCMM_ERROR CMM_StartNode( tCMM_HANDLE hBoard, BYTE NodeNo );`

**Parameters:**

Parameter	Dir.	Explanation
hBoard	(in)	Handle of the CAN board
NodeNo	(in)	CANopen node-ID of a network node, 0 for the complete network

**Return values:**

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_NOT_SENT	Command interface allocated, command not transmitted, another attempt necessary
CMMERR_NOT_CONFIGURED	CANopen Manager was not configured
CMMERR_NO_OBJECTS	No data in communication queues
CMMERR_COMMTIMEOUT	Timeout during communication with firmware
CMMERR_CCI_INST_ERR	CCI installation error
CMMERR_GENERAL_ERR	General error
CMMERR_NOT_AUTHORIZED	CANopen Manager is not local CANopen master
CMMERR_INVALID_PARAM	Invalid parameter
CMMERR_STATE_ERR	Command not allowed in current state of the CANopen Manager

### 11.3.4 CMM\_StopNode

**Description:** NMT function. This function is used to set a CANopen node (also of the CANopen Manager itself, in so far as its node number is given) or of the complete CANopen network in the **Stopped** state using the NMT command **Stop Remote Node**.

**Prototype:** `tCMM_ERROR CMM_StopNode( tCMM_HANDLE hBoard, BYTE NodeNo );`

**Parameters:**

Parameter	Dir.	Explanation
HBoard	(in)	Handle of the CAN board
NodeNo	(in)	CANopen node-ID of a network node, 0 for the complete network

**Return values:**

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_NOT_SENT	Command interface allocated, command not transmitted, another attempt necessary
CMMERR_NOT_CONFIGURED	CANopen Manager was not configured
CMMERR_NO_OBJECTS	No data in communication queues
CMMERR_COMMTIMEOUT	Timeout during communication with firmware
CMMERR_CCI_INST_ERR	CCI installation error
CMMERR_GENERAL_ERR	General error
CMMERR_NOT_AUTHORIZED	CANopen Manager is not local CANopen master
CMMERR_INVALID_PARAM	Invalid parameter
CMMERR_STATE_ERR	Command not allowed in current state of the CANopen Manager

### 11.3.5 CMM\_EnterPreOp

**Description:** NMT function. This function is used to set a CANopen node (also of the CANopen Manager itself, in so far as its node number is given) or of the complete CANopen network to the **Pre-operational** state using the NMT command **Enter Pre-Operational**.

**Prototype:** `tCMM_ERROR CMM_EnterPreOp( tCMM_HANDLE hBoard, BYTE NodeNo );`

**Parameters:**

Parameter	Dir.	Explanation
hBoard	(in)	Handle of the CAN board
NodeNo	(in)	CANopen node-ID of a network node, 0 for the complete network

**Return values:**

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_NOT_SENT	Command interface allocated, command not transmitted, another attempt necessary
CMMERR_NOT_CONFIGURED	CANopen Manager was not configured
CMMERR_NO_OBJECTS	No data in communication queues
CMMERR_COMMTIMEOUT	Timeout during communication with firmware
CMMERR_CCI_INST_ERR	CCI installation error
CMMERR_GENERAL_ERR	General error
CMMERR_NOT_AUTHORIZED	CANopen Manager is not local CANopen master
CMMERR_INVALID_PARAM	Invalid parameter
CMMERR_STATE_ERR	Command not allowed in current state of the CANopen Manager

### 11.3.6 CMM\_ResetComm

**Description:** NMT function. This function is used to reset the communication of a CANopen node (also of the CANopen Manager itself, in so far as its node number is given) or of the complete CANopen network using the NMT command **Reset Communication**.

For the CANopen Manager as the target node this command corresponds to the function call `CMM_InitFirmware()` with `InitMode == COP_k_RESETCOM`.

**Prototype:** `tCMM_ERROR CMM_ResetComm( tCMM_HANDLE hBoard, BYTE NodeNo );`

**Parameters:**

Parameter	Dir.	Explanation
hBoard	(in)	Handle of the CAN board
NodeNo	(in)	CANopen node-ID of a network node, 0 for the complete network

**Return values:**

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_NOT_SENT	Command interface allocated, command not transmitted, another attempt necessary
CMMERR_NOT_CONFIGURED	CANopen Manager was not configured
CMMERR_NO_OBJECTS	No data in communication queues
CMMERR_COMMTIMEOUT	Timeout during communication with firmware
CMMERR_CCI_INST_ERR	CCI installation error
CMMERR_GENERAL_ERR	General error
CMMERR_NOT_AUTHORIZED	CANopen Manager is not local CANopen master
CMMERR_INVALID_PARAM	Invalid parameter
CMMERR_STATE_ERR	Command not allowed in current state of the CANopen Manager

### 11.3.7 CMM\_ResetNode

**Description:** NMT function. This function is used to reset a CANopen node (also of the CANopen Manager itself, in so far as its node number is given) or of the complete CANopen network using the NMT command **Reset Node**.  
For the CANopen Manager as the target node, this command corresponds to the function call `CMM_InitFirmware()` with `InitMode == COP_k_RESETNODE`.

**Prototype:** `tCMM_ERROR CMM_ResetNode( tCMM_HANDLE hBoard, BYTE NodeNo );`

**Parameters:**

Parameter	Dir.	Explanation
HBoard	(in)	Handle of the CAN board
NodeNo	(in)	CANopen node-ID of a network node, 0 for the complete network

**Return values:**

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_NOT_SENT	Command interface allocated, command not transmitted, another attempt necessary
CMMERR_NOT_CONFIGURED	CANopen Manager was not configured
CMMERR_NO_OBJECTS	No data in communication queues
CMMERR_COMMTIMEOUT	Timeout during communication with firmware
CMMERR_CCI_INST_ERR	CCI installation error
CMMERR_GENERAL_ERR	General error
CMMERR_NOT_AUTHORIZED	CANopen Manager is not local CANopen master
CMMERR_INVALID_PARAM	Invalid parameter
CMMERR_STATE_ERR	Command not allowed in current state of the CANopen Manager

## 11.4 Object Dictionary and SDO related Functions

### 11.4.1 CMM\_CreateODentry

**Description:** The function `CMM_CreateODentry()` generates an object dictionary entry in the local object dictionary of the CANopen Manager.

This function is only available in the *Reset* state of the CANopen Manager. If the object dictionary entries thus generated are to be retained even after a reset of the CANopen Manager, the **Store Parameters** command is to be carried out by means of SDO write access to the local object dictionary entry [1010].

**Prototype:**

```

tCMM_ERROR CMM_CreateODentry(
    tCMM_HANDLE    hBoard,
    WORD           Idx,
    BYTE           Subidx,
    eCMM_DATATYPE  Datatype,
    eCMM_ACCESSTYPE Accesstype,
    eCMM_PDOMAPPING Mappable,
    BYTE           InitialValue[8],
    WORD           PIOffset );
    
```

**Parameters:**

Parameter	Dir.	Explanation
HBoard	(in)	Handle of the CAN board
Idx	(in)	Object dictionary index of the new entry. Values are possible in the range [2000]..[9FFF], except the range reserved for the CANopen Manager [5F00]..[5FFF]
Subidx	(in)	Sub index of the new object dictionary entry
Datatype	(in)	CANopen data type, simultaneously defines the number of bytes the value allocates in the PI: DATATYPE_INTEGER8 DATATYPE_INTEGER16 DATATYPE_INTEGER32 DATATYPE_UNSIGNED8 DATATYPE_UNSIGNED16 DATATYPE_UNSIGNED32 DATATYPE_UNSIGNED64 DATATYPE_INTEGER64

## Individual Functions of the API-DLL

---

AccessType	(in)	Access type, defines the data direction and the PI: <b>ACCESSTYPE_RWW:</b> Value is stored in the PI input and can be mapped in RPDOs, it can be read and written via SDO <b>ACCESSTYPE_RO</b> Value is stored in the PI output and can be mapped in TPDOs, it can only be read via SDO
Mappable	(in)	Defines whether the object should be mappable in a PDO or not: <b>PDOMAPPING_UNSUPPORTED:</b> Value cannot be mapped <b>PDOMAPPING_SUPPORTED:</b> Value can be mapped in a PDO
InitialValue[8]	(in)	Default value of the new object dictionary entry
PIOffset	(in)	Byte offset in PI input or PI output

### Return values:

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_NOT_SENT	Command interface busy, command not transmitted, another attempt necessary
CMMERR_NOT_CONFIGURED	CANopen Manager was not configured
CMMERR_NO_OBJECTS	No data in communication queues
CMMERR_COMMTIMEOUT	Timeout during communication with firmware
CMMERR_CCI_INST_ERR	CCI installation error
CMMERR_GENERAL_ERR	Object dictionary entry already created
CMMERR_MAX_ENTRIES_REACHED	Maximum number of dynamically created object dictionary entries exceeded
CMMERR_INVALID_PARAM	Invalid parameter value
CMMERR_STATE_ERR	CANopen Manager in incorrect state



### 11.4.2 CMM\_ReadSDO

**Description:** CMM\_ReadSDO() carries out an SDO upload from a network node. If the stated size of the data buffer is smaller than the number of data bytes of the object read out, the data buffer is filled up to its size and the actual size of the read object returned in rxlen. rxlen will always be overwritten inside the function.

**Prototype:**

```

tCMM_ERROR CMM_ReadSDO( tCMM_HANDLE    hBoard,
                          BYTE             NodeNo,
                          BYTE             SdoNo,
                          eCMM_SDOMODE    Mode,
                          WORD            Idx,
                          BYTE            SubIdx,
                          DWORD*         rxlen,
                          BYTE*         rxdata,
                          DWORD*         pAbortcode );
```

**Parameters:**

Parameter	Dir.	Explanation
HBoard	(in)	Handle of the CAN board
NodeNo	(in)	CANopen node-ID of the addressed slave
SdoNo	(in)	CMM_DEFAULT_SDO
Mode	(in)	Defines the SDO transmission type: SDOMODE_SEGMENTED: Expedited or segmented SDO transfer
Idx	(in)	Object dictionary index of the value to be read
SubIdx	(in)	Sub index of the value to be read
Rxlen	(in/out)	in: size of the data buffer for the read data in bytes out: number of data bytes read
Rxdata	(out)	Return of received data
pAbortcode	(out)	Return of the SDO abort code

**Return values:**

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_GENERAL_ERR	General error
CMMERR_NOT_AUTHORIZED	CANopen Manager not local Master
CMMERR_INVALID_PARAM	Invalid parameter value
CMMERR_NOT_CONFIGURED	CANopen Manager was not configured
CMMERR_STATE_ERR	command not allowed in current state
CMMERR_MEM_ALLOC_ERR	An operating system object that is required for the SDO access was not provided
CMMERR_NOT_SENT	Command interface busy, command not transmitted, another attempt necessary
CMMERR_COMMTIMEOUT	Timeout during communication with firmware

## Individual Functions of the API-DLL

---

CMMERR_SDO_INUSE	SDO transfer in progress, further SDO transfer not possible
CMMERR_SDO_TIMEOUT	Timeout in SDO communication
CMMERR_SDO_STOPPED	CANopen Manager is in <b>Stopped</b> state
CMMERR_SDO_ABORT_CLIENT	SDO transfer aborted by Client
CMMERR_SDO_ABORT_SERVER	SDO transfer aborted by Server

### 11.4.3 CMM\_WriteSDO

**Description:** CMM\_WriteSDO() carries out an SDO download to a network node.

**Prototype:**

```

tCMM_ERROR CMM_WriteSDO( tCMM_HANDLE   hBoard,
                          BYTE          NodeNo,
                          BYTE          SdoNo,
                          eCMM_SDOMODE Mode,
                          WORD          Idx,
                          BYTE          SubIdx,
                          DWORD         TxLen,
                          BYTE*         TxData,
                          DWORD*       pAbortCode );
    
```

**Parameters:**

Parameter	Dir.	Explanation
HBoard	(in)	Handle of the CAN board
NodeNo	(in)	CANopen node-ID of the addressed slave
SdoNo	(in)	CMM_DEFAULT_SDO
Mode	(in)	Defines SDO transmission type: SDOMODE_SEGMENTED: Expedited or segmented SDO transfer
Idx	(in)	Object dictionary index of the value to be read
SubIdx	(in)	Sub index of the value to be written
TxLen	(in)	Number of data bytes to be transmitted
TxData	(in)	Data to be transmitted
pAbortCode	(out)	Return of the SDO abort code

**Return values:**

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_GENERAL_ERR	General error
CMMERR_NOT_AUTHORIZED	CANopen Manager not local Master
CMMERR_INVALID_PARAM	Invalid parameter value
CMMERR_NOT_CONFIGURED	CANopen Manager was not configured
CMMERR_STATE_ERR	command not allowed in current state
CMMERR_MEM_ALLOC_ERR	An operating system object that is required for the SDO access was not provided
CMMERR_NOT_SENT	Command interface busy, command not transmitted, another attempt necessary
CMMERR_COMMTIMEOUT	Timeout during communication with firmware
CMMERR_SDO_INUSE	SDO transfer in progress, further SDO transfer not possible
CMMERR_SDO_TIMEOUT	Timeout in SDO communication
CMMERR_SDO_STOPPED	CANopen Manager is in <b>Stopped</b> state
CMMERR_SDO_ABORT_CLIENT	SDO transfer aborted by Client
CMMERR_SDO_ABORT_SERVER	SDO transfer aborted by Server

### 11.4.4 CMM\_ReadLocSDO

**Description:** CMM\_ReadLocSDO() carries out an SDO upload from the local object dictionary of the CANopen Manager.  
If the stated size of the data buffer is smaller than the number of data bytes of the object read out, the data buffer is filled up to its size and the actual size of the read object returned in rxlen.

**Prototype:**

```
tCMM_ERROR CMM_ReadLocSDO(  
    tCMM_HANDLE hBoard,  
    WORD Idx,  
    BYTE Subidx,  
    DWORD* rxlen,  
    BYTE* rxdata,  
    DWORD* pAbortcode );
```

**Parameters:**

Parameter	Dir.	Explanation
HBoard	(in)	Handle of the CAN board
Idx	(in)	Object dictionary index of the value to be read
Subidx	(in)	Sub index of the value to be read
Rxlen	(in/out)	in: size of the data buffer for the read data in bytes out:: number of data bytes read
Rxdata	(out)	Return of the received data
PAbortcode	(out)	Return of the SDO abort code

**Return values:**

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_GENERAL_ERR	General error
CMMERR_INVALID_PARAM	Invalid parameter value
CMMERR_NOT_CONFIGURED	CANopen Manager was not configured
CMMERR_STATE_ERR	Command not allowed in current state
CMMERR_MEM_ALLOC_ERR	An operating system object that is required for the SDO access was not provided
CMMERR_NOT_SENT	Command interface allocated, command not transmitted, another attempt necessary
CMMERR_COMMTIMEOUT	Timeout during communication with firmware
CMMERR_SDO_INUSE	SDO transfer in progress, further SDO transfer not possible
CMMERR_SDO_TIMEOUT	Timeout in SDO communication
CMMERR_SDO_STOPPED	CANopen Manager is in <b>Stopped</b> state
CMMERR_SDO_ABORT_SERVER	Read access to object dictionary of Manager has resulted in an abort

### 11.4.5 CMM\_WriteLocSDO

**Description:** CMM\_WriteLocSDO() carries out an SDO download to the local object dictionary of the CANopen Manager.

**Prototype:**

```

tCMM_ERROR CMM_WriteLocSDO(
    tCMM_HANDLE   hBoard,
    WORD          Idx,
    BYTE         SubIdx,
    DWORD        txLen,
    BYTE*       txdata,
    DWORD*      pAbortcode );
    
```

**Parameters:**

Parameter	Dir.	Explanation
hBoard	(in)	Handle of the CAN board
Idx	(in)	Object dictionary index of the value to be read
SubIdx	(in)	Sub index of the value to be written
txLen	(in)	Number of data bytes to be transmitted
txdata	(in)	Data to be transmitted
pAbortcode	(out)	Return of the SDO abort code

**Return values:**

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_GENERAL_ERR	General error
CMMERR_INVALID_PARAM	Invalid parameter value
CMMERR_NOT_CONFIGURED	CANopen Manager was not configured
CMMERR_STATE_ERR	Command not allowed in current state
CMMERR_MEM_ALLOC_ERR	An operating system object that is required for the SDO access was not provided
CMMERR_NOT_SENT	Command interface busy, command not transmitted, another attempt necessary
CMMERR_COMMTIMEOUT	Timeout during communication with firmware
CMMERR_SDO_INUSE	SDO transfer in progress, further SDO transfer not possible
CMMERR_SDO_TIMEOUT	Timeout in SDO communication
CMMERR_SDO_STOPPED	CANopen Manager is in <b>Stopped</b> state
CMMERR_SDO_ABORT_SERVER	Write access to object dictionary of Manager has resulted in an abort

### 11.4.6 CMM\_ImportCDC

**Description:** Use this function to import a Concise DCF to Manager. The Concise DCF values will be written to the Manager's local Object Dictionary.

**Prototype:** `tCMM_ERROR CMM_ImportCDC( tCMM_HANDLE hBoard, wchar_t* CDCFile, WORD* pIdx, BYTE* pSubidx, DWORD* pAbortcode );`

**Parameters:**

Parameter	Dir.	Explanation
hBoard	(in)	Handle of the CAN board
CDCFile	(in)	full absolute filename and path to a Concise DCF
pIdx	(out)	In case of CMMERR_SDO_ABORT_SERVER, the index of the aborted SDO transfer will be delivered in this optional argument
pSubidx	(out)	In case of CMMERR_SDO_ABORT_SERVER, the sub-index of the aborted SDO transfer will be delivered in this optional argument
pAbortcode	(out)	Return of the SDO abort code, in case of CMMERR_SDO_ABORT_SERVER, the abort code of the aborted SDO transfer will be delivered in this optional argument

**Return values:**

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_INVALID_PARAM	Invalid parameter value
CMMERR_CDC_CORRUPT	Concise DCF import failed. File may be corrupt
CMMERR_NOT_CONFIGURED	CANopen Manager was not configured
CMMERR_STATE_ERR	Command not allowed in current state
CMMERR_MEM_ALLOC_ERR	An operating system object that is required for the SDO access was not provided
CMMERR_NOT_SENT	Command interface busy, command not transmitted, another attempt necessary
CMMERR_COMMTIMEOUT	Timeout during communication with firmware
CMMERR_SDO_INUSE	SDO transfer in progress, further SDO transfer not possible
CMMERR_SDO_TIMEOUT	Timeout in SDO communication
CMMERR_SDO_STOPPED	CANopen Manager is in <b>Stopped</b> state
CMMERR_SDO_ABORT_SERVER	Write access to object dictionary of Manager has resulted in an abort

## 11.5 Process image-related functions

### 11.5.1 CMM\_FormPILUT

**Description:** Generate the internal lookup table for all the Process Image entries. This is possible in AutoConfiguration Mode only.

Attention: You must call this function right after the boot-up in AutoConfiguration Mode has finished successfully, i.e. `LOWBYTE(wMasterManagerState) == GETPI_INFO`.

The individual process image entry then can be read out using the function `CMM_GetPIdescr()`.

**Prototype:** `tCMM_ERROR CMM_FormPILUT( tCMM_HANDLE hBoard );`

**Parameters:**

Parameter	Dir.	Explanation
hBoard	(in)	Handle of the CAN board

**Return values:**

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_NOT_SENT	Message not sent, try again
CMMERR_NOT_CONFIGURED	CANopen Manager was not configured
CMMERR_COMMTIMEOUT	Timeout in communication PC to $\mu$ C
CMMERR_CCI_INST_ERR	CCI installation error (internal)
CMMERR_NO_OBJECTS	No more entries
CMMERR_NOT_AUTHORIZED	CANopen Manager not local master
CMMERR_STATE_ERR	Command not allowed in current state
CMMERR_PI_ERR	Process Image inconsistencies
CMMERR_INVALID_CMD	Command code is not supported

### 11.5.2 CMM\_GetPIdescr

**Description:** This function supplies the properties of an individual process image entry. This function can only be called in AutoConfiguration Mode.

First the function `CMM_StartAutoConfig()` must be called and the boot-up process must be terminated. Second the function `CMM_FormPILUT()` must be called once. Then the function `CMM_GetPIdescr()` can be called for each process image entry.

The calls must first be carried out for PI input and then for PI output until the function returns `MMERR_NO_OBJECTS` as an indication of the end.

**Prototype:**

```
tCMM_ERROR CMM_GetPIdescr(  
                                tCMM_HANDLE hBoard,  
                                eCMM_PITYPE Pitype,  
                                WORD* pPIoffset,  
                                BYTE* pLength,  
                                BYTE* pPdoNo,  
                                BYTE* pNodeNo,  
                                WORD* pRemoteIdx,  
                                BYTE* pRemoteSubIdx,  
                                BYTE* pRemotePdoNo,  
                                WORD* pProfile,  
                                BYTE* pProductCode );
```

**Parameters:**

Parameter	Dir.	Explanation
<code>hBoard</code>	(in)	Handle of the CAN board
<code>Pitype</code>	(in)	Selection PI input or PI output: <code>PITYPE_INPUTS</code> : PI input <code>PITYPE_OUTPUTS</code> : PI output
<code>pPIoffset</code>	(out)	Start address of the entry in the PI (byte offset)
<code>pLength</code>	(out)	Length of the entry in bytes (max. 8 bytes)
<code>pPdoNo</code>	(out)	PDO number of the corresponding RPDO/TPDO, zero based
<code>pNodeNo</code>	(out)	CANopen node-ID of the network node to which the entry is mapped
<code>pRemoteIdx</code>	(out)	Object dictionary index of remote object to which the entry is mapped
<code>pRemoteSubIdx</code>	(out)	Object dictionary sub-index of remote object to which the entry is mapped
<code>pRemotePdoNo</code>	(out)	PDO number of the corresponding remote RPDO/TPDO, zero based
<code>pProfile</code>	(out)	Profile of the CANopen node to which the entry is mapped
<code>pProductCode</code>	(out)	Product code of the CANopen node to which the entry is mapped



### Return values:

<b>Return value</b>	<b>Description</b>
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_INVALID_PARAM	Invalid parameter value
CMMERR_NO_OBJECTS	No object in the queue

### 11.5.3 CMM\_GetPI

**Description:** The function `CMM_GetPI()` supplies an image of the current PI input or PI output. The process image is copied into the stipulated buffer `pPI`.

**Prototype:**

```
tCMM_ERROR CMM_GetPI( tCMM_HANDLE hBoard,
                      eCMM_PITYPE  Pitype,
                      DWORD*       pLength,
                      BYTE*        pPI,
                      DWORD*       pTimestamp );
```

**Parameters:**

Parameter	Dir.	Explanation
<code>hBoard</code>	(in)	Handle of the CAN board
<code>Pitype</code>	(in)	Selection PI input or PI output: PITYPE_INPUTS: PI input PITYPE_OUTPUTS: PI output
<code>pLength</code>	(in/out)	in: size of the buffer for the data to be read out: number of bytes currently used
<code>pPI</code>	(out)	Return of the PI data
<code>pTimestamp</code>	(out)	timestamp value of PII Indicates the specific moment when the PII had been updated. Use this optional value to find the belonging entries in the PII-RPDO queue (see also function <code>CMM_GetPIIRPDOOno()</code> )

**Return values:**

Return value	Description
<code>CMMERR_OK</code>	Success
<code>CMMERR_INVALID_HANDLE</code>	Invalid board handle
<code>CMMERR_NOT_CONFIGURED</code>	CANopen Manager was not configured
<code>CMMERR_INVALID_PARAM</code>	Invalid parameter value
<code>CMMERR_PI_LOCKED</code>	Process Image currently accessed by firmware, try again

### 11.5.4 CMM\_GetPIentry

**Description:** CMM\_GetPIentry() supplies an excerpt from the PI input or PI output. The contents of the selected excerpt are copied into the stipulated buffer pPIentry.

**Prototype:**

```

tCMM_ERROR CMM_GetPIentry( tCMM_HANDLE hBoard,
                             eCMM_PITYPE Pitype,
                             DWORD dwPIoffset,
                             DWORD* pLength,
                             BYTE* pPIentry,
                             DWORD* pTimestamp );
```

**Parameters:**

Parameter	Dir.	Explanation
hBoard	(in)	Handle of the CAN board
Pitype	(in)	Selection PI input or PI output: PITYPE_INPUTS: PI input PITYPE_OUTPUTS: PI output
dwPIoffset	(in)	Byte offset of the entry to be read
pLength	(in/out)	in: size of the buffer for the data to be read out: number of bytes currently used
pPIentry	(out)	Return of the PI subset
pTimestamp	(out)	timestamp value of PII Indicates the specific moment when the PII had been updated. Use this optional value to find the belonging entries in the PII-RPDO queue (see also function CMM_GetPIIRPDOno())

**Return values:**

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_NOT_CONFIGURED	CANopen Manager was not configured
CMMERR_INVALID_PARAM	Invalid parameter value
CMMERR_PI_LOCKED	Process Image currently accessed by firmware

### 11.5.5 CMM\_GetPIIvalue

**Description:** CMM\_GetPIIvalue() returns the value of the given remote object from the binary Process Image Inputs of CANopen Manager.

This functions is available only in AutoConfiguration Mode. It utilizes the internal Process Image Look-Up Table (PILUT) to locate the remote object in the Process Image Inputs. Thus it is indispensable to call CMM\_FormPILUT() once after AutoConfiguration Mode finished successfully.

Note: Since the lookup will be performed every time this function is called, it is only suitable for slow request intervals resp. few requests per interval. For higher performance, call CMM\_GetPIentry(). If maximum performance is required, call CMM\_GetPI() and resolve the PI contents in the client application.

**Prototype:**

```
tCMM_ERROR CMM_GetPIIvalue( tCMM_HANDLE hBoard,
                             BYTE         NodeNo,
                             WORD         Idx,
                             BYTE         Subidx,
                             DWORD*      pLength,
                             BYTE*       pPIvalue);
```

**Parameters:**

Parameter	Dir.	Explanation
hBoard	(in)	Handle of the CAN board
NodeNo	(in)	Node number of the remote node
Idx	(in)	Index of remote object
Subidx	(in)	Sub index of remote object
pLength	(in/out)	Size of Process Image value buffer / actually used bytes
pPIvalue	(out)	Buffer for Process Image value

**Return values:**

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_NO_SUCH_OBJECT	Given object not found in PILUT
CMMERR_NOT_CONFIGURED	CANopen Manager was not configured
CMMERR_INVALID_PARAM	Invalid parameter value
CMMERR_PI_LOCKED	Process Image currently accessed by firmware

### 11.5.6 CMM\_PutPIO

**Description:** CMM\_PutPIO() writes the complete PI output with new data.

**Prototype:** `tCMM_ERROR CMM_PutPIO( tCMM_HANDLE hBoard, DWORD dwLength, BYTE* pPI );`

**Parameters:**

Parameter	Dir.	Explanation
hBoard	(in)	Handle of the CAN board
dwLength	(in)	Number of bytes to be written
pPI	(in)	PI output data

**Return values:**

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_NOT_CONFIGURED	CANopen Manager was not configured
CMMERR_INVALID_PARAM	Invalid parameter value
CMMERR_PI_LOCKED	Process Image currently accessed by firmware

### 11.5.7 CMM\_PutPIOentry

**Description:** CMM\_PutPIOentry() writes an excerpt of the PI output with new data.

**Prototype:** `tCMM_ERROR CMM_PutPIOentry( tCMM_HANDLE hBoard, DWORD dwPIOffset, DWORD dwLength, BYTE* pPIentry );`

**Parameters:**

Parameter	Dir.	Explanation
hBoard	(in)	Handle of the CAN board
dwPIOffset	(in)	Offset of the PI output entry in bytes
dwLength	(in)	Number of bytes to be written
pPI	(in)	PI output data

**Return values:**

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_NOT_CONFIGURED	CANopen Manager was not configured
CMMERR_INVALID_PARAM	Invalid parameter value
CMMERR_PI_LOCKED	Process Image currently accessed by firmware

### 11.5.8 CMM\_PutPIOvalue

**Description:** CMM\_PutPIOvalue() writes a value for the specified remote object to the binary Process Image Outputs of the CANopen Manager.

This functions is available only in AutoConfiguration Mode. It utilises the internal Process Image Look-Up Table (PILUT) to locate the remote object in the Process Image Outputs. Thus it is indispensable to call CMM\_FormPILUT() once after AutoConfiguration Mode finished successfully.

Note: Since the lookup will be performed every time this function is called, it is only suitable for slow request intervals respectively few requests per interval. For higher performance, call CMM\_PutPIOentry(). If maximum performance is required, place the value in an image of the PI contents in the client application and call CMM\_PutPIO().

**Prototype:**

```

tCMM_ERROR CMM_PutPIOvalue( tCMM_HANDLE hBoard,
                             BYTE         NodeNo,
                             WORD         Idx,
                             BYTE         SubIdx,
                             BYTE*        pPIvalue );
```

**Parameters:**

Parameter	Dir.	Explanation
hBoard	(in)	Handle of the CAN board
NodeNo	(in)	Node number of the remote node
Idx	(in)	Index of remote object
SubIdx	(in)	Sub index of remote object
pPIvalue	(in)	Process Image value

**Return values:**

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_NO_SUCH_OBJECT	Given object not found in PILUT
CMMERR_NOT_CONFIGURED	CANopen Manager was not configured
CMMERR_INVALID_PARAM	Invalid parameter value
CMMERR_PI_LOCKED	Process Image currently accessed by firmware

### 11.5.9 CMM\_GetPIIRPDOno

**Description:** Fetch an entry from the Process Image Input Receive PDO queue.

**Prototype:** `tCMM_ERROR CMM_GetPIIRPDOno(  
tCMM_HANDLE hBoard,  
DWORD* pTimestamp,  
BYTE* RPDOno );`

**Parameters:**

Parameter	Dir.	Explanation
hBoard	(in)	Handle of the CAN board
pTimestamp	(out)	timestamp value of RPDO Indicates the specific moment when the RPDO data had been entered in the PII. Use this value to find out which RPDO belongs to your Process Image snapshot (see also function CMM_GetPI)s
RPDOno	(out)	zero based number of a CANopen Manager's RPDO, the separator code 0xFF indicates a moment at which the Process Image had been updated

**Return values:**

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_INVALID_PARAM	Invalid parameter value
CMMERR_NO_OBJECTS	No objects in queue
CMMERR_CCI_INST_ERR	CCI installation error



### 11.5.10 CMM\_TriggerPIOoffset

**Description:** Trigger the TPDO(s) that contain given output Process Image offset. Offset given must be the start address of a mapped object.

This function shall be used when PDOs shall be transmitted again although the content of the mapped data has not changed.

Note: When new (changed) data are written into the Process Image Output the assigned TPDOs will be transmitted automatically without calling this function.

If the CANopen Manager firmware detects an error on execution the command an event CMM\_NOTI\_KIND\_TRIGGERTPDOQUEUE will be posted into the event queue.

**Prototype:**

```
tCMM_ERROR CMM_TriggerPIOoffset(
                                tCMM_HANDLE hBoard,
                                DWORD dwPIOoffset );
```

**Parameters:**

Parameter	Dir.	Explanation
hBoard	(in)	Handle of the CAN board
dwPIOoffset	(in)	Start address of a mapped object relative to beginning of output Process Image

**Return values:**

Return value	Description
CMMERR_OK	Success
CMMERR_INVALID_HANDLE	Invalid board handle
CMMERR_NOT_SENT	Message not sent, try again
CMMERR_INVALID_PARAM	Invalid parameter value



Folder	File name	Meaning
lib\MSVC\	XatCMM20.lib	Library file for Microsoft Visual C++ Lib-file of the CANopen Manager API

Folder	File name	Meaning
lib\x64\	XatCMM20.lib	64bit library file for Microsoft Visual C++ Lib-file of the CANopen Manager API

Folder	File name	Meaning
<windows>	XatCMM20.dll	Windows system folder CANopen Manager API (Release-Version)

Folder	File name	Meaning
Samples\	XatCMM.h XatCMMdef.h XatBrds.h  vcguid.h	Sample projects and common Header files C main header of the CANopen Manager API Definitions of the data types Definitions of all IXXAT Hardware interface boards (CAN boards VCI 2) Definitions of all IXXAT Hardware interface boards (CAN boards VCI3)

Folder	File name	Meaning
Samples\Tutorial\	Tutorial.cpp Tutorial.exe XatCMMutil.cpp XatCMMutil.h Tutorial.dsp Tutorial.sln Tutorial.vcproj	Tutorial application as described in chapter 4 for Microsoft Visual Studio Source file of sample tutorial application 32bit executable tutorial application, release build Common utility functions for conversion Header of Common utility functions for conversion Visual Studio 6 project file Visual Studio 2005 Solution Visual C++ 8.0 project file

Folder	File name	Meaning
Samples\Tutorial\x64\Release\	Tutorial.exe	Tutorial application as described in chapter 4 for Microsoft Visual Studio 64bit executable tutorial application, release build

## Appendix – Scope of Delivery

Folder	File name	Meaning
Samples\XatCMMdiag\		Example application for use with Microsoft Visual C++
	XATCMMdiag.cpp	Source file of sample program for usage of API
	XATCMMdiag.h	Header of sample program for usage of API
	XATCMMdiag.exe	32bit executable example application, release build
	XATCMMdiag.dsp	Visual Studio 6 project file
	XATCMMdiag.sln	Visual Studio 2005 Solution
	XATCMMdiag.vcproj	Visual C++ 8.0 project file
	XatCMMPDopoll.cpp	Poll thread for cyclic process image readout
	XatCMMPDopoll.h	Header of Poll thread for cyclic process image readout
	XatCMMutil.cpp	Common utility functions for conversion
	XatCMMutil.h	Header of Common utility functions for conversion
	MegaNode@10.cdc	Example concise DCF

Folder	File name	Meaning
Samples\XatCMMdiag\x64\Release\		Example application for use with Microsoft Visual C++
	XATCMMdiag.exe	64bit executable example application, release build

Folder	File name	Meaning
Tools\		Utility programs
	Xcflash.exe	VCI2 Flash programmer for iPC-I XC16/PCI CAN interface boards
	ucii161f.H86	VCI2 UCI/VCI flash firmware for iPC-I XC16/PCI CAN interface board
	XATCMMFL.H86	VCI2 CANopen Manager flash firmware for iPC-I XC16/PCI CAN interface board
	XatCMMFL3.H86	VCI3 CANopen Manager flash firmware for iPC-I XC16/PCI CAN interface board