

CANopen User Manual

Software Manual

Edition September 2015

In this manual are descriptions for copyrighted products, which are not explicitly indicated as such. The absence of the trademark ® and copyright © symbols does not infer that a product is not protected. Additionally, registered patents and trademarks are similarly not expressly indicated in this manual

The information in this document has been carefully checked and is believed to be entirely reliable. However, SYS TEC electronic GmbH assumes no responsibility for any inaccuracies. SYS TEC electronic GmbH neither gives any guarantee nor accepts any liability whatsoever for consequential damages resulting from the use of this manual or its associated product. SYS TEC electronic GmbH reserves the right to alter the information contained herein without prior notification and accepts no responsibility for any damages, which might result.

Additionally, SYS TEC electronic GmbH offers no guarantee nor accepts any liability for damages arising from the improper usage or improper installation of the hardware or software. SYS TEC electronic GmbH further reserves the right to alter the layout and/or design of the hardware without prior notification and accepts no liability for doing so.

© Copyright 2015 SYS TEC electronic GmbH All Rights - including those of translation, reprint, broadcast, photomechanical or similar reproduction and storage or processing in computer systems, in whole or in part - are reserved. No reproduction may occur without the express written consent from SYS TEC electronic GmbH.

	Directly	Your local distributor
Address:	SYS TEC electronic GmbH Am Windrad 2 D-08468 Heinsdorfergrund GERMANY	You find a list of our distributors under http://www.systec-electronic.com/distributors
Ordering Information:	+49 (0) 3765 / 38600-2110 info@systec-electronic.com	
Technical Support:	+49 (0) 3765 / 38600-2140 support@systec-electronic.com	
Fax:	+49 (0) 3765 / 38600-4100	
Web Site:	http://www.systec-electronic.com	

14th Edition September 2015

Table of Contents
PREFACE 7

1	CANopen Fundamentals	9
1.1	What is CANopen?	10
1.2	Communication objects (COB).....	13
1.2.1	Process data object (PDO)	13
1.2.2	Service data object (SDO)	19
1.2.1	Synchronization object (SYNC).....	21
1.2.2	Time stamp object (TIME).....	21
1.2.3	Emergency object (EMCY).....	21
1.2.4	Layer setting service (LSS).....	22
1.3	Network Management (NMT).....	25
1.4	CANopen communication profile.....	29
1.5	Transmission protocols	30
1.6	Object dictionary (OD)	30
1.7	Error handling and reporting	31
1.8	Telegram table (pre-defined connection set).....	32
2	CANopen User Layer	33
2.1	Software structure	33
2.1.1	CANopen stack.....	34
2.1.2	Hardware-specific layer.....	36
2.1.3	Application-specific layer.....	36
2.2	Directory structure.....	39
2.3	Data structures.....	40
2.4	Object dictionary	43
2.4.1	Example object dictionary	43
2.5	Instanceability of the CANopen layer	45
2.5.1	Using the instance handle.....	46
2.5.2	Using instance pointers.....	46
2.6	Hints for creating an application.....	47
2.6.1	Selecting the required modules and configuration	47
2.6.2	Sequence of a CANopen application	49
2.7	Description of CCM layer functions	60
2.7.1	Description of module CcmMain	60
2.7.2	Description of module CcmSdoc.....	78
2.7.3	Description of module CcmDfPdo	89
2.7.4	Description of module CcmObj	91
2.7.5	Description of module CcmLgs	93
2.7.6	Description of module CcmStore	95
2.7.7	Description of module CcmNmtm and CcmNmtm.....	103
2.7.8	Description of module CcmSnPdo	109
2.7.9	Description of module CcmSync	109
2.7.10	Description of module CcmSyncEx.....	113
2.7.11	Description of module CcmEmcc	113
2.7.12	Description of module CcmEmcp	117
2.7.13	Description of module CcmHbc.....	121
2.7.14	Description of module CcmHbp	125
2.7.15	Description of module TgtCav and CavFile.....	125
2.7.16	Description of module CcmBoot.....	134
2.7.17	Description of module CcmFloat	135
2.7.18	Description of module CcmStPdo	136
2.7.19	Description of module Ccm303	138
2.7.20	Description of module CcmLss	144
2.7.21	Communication parameters and process variables	153

2.8	Description of the CANopen stack functions	155
2.8.1	Description of module SDOS	155
2.8.2	Description of module SDOC	171
2.8.3	Description of module PDO	186
2.8.4	Description of module PDOSTC.....	198
2.8.5	Description of module OBD.....	200
2.8.6	Description of module COB	212
2.8.7	Description of module NMT	217
2.8.8	Description of module NMTS	219
2.8.9	Description of module NMTM.....	221
2.8.10	Description of module EMCC	225
2.8.11	EMCP module	228
2.8.12	Description of module HBC	231
2.8.13	Description of module HBP	234
2.9	Add-on modules for the CANopen protocol stack	237
2.9.1	Description of module MPDO.....	237
2.9.2	Description of module CcmMPdo.....	239
2.10	Meaning of return values and abort codes	241
2.10.1	CANopen return codes.....	241
2.10.2	SDO abort codes.....	248
2.10.3	Emergency error codes.....	249
2.11	Configuration and Scaling	250
2.11.1	Configuration of the CANopen stack	250
2.11.2	Configuration of the Object Dictionary.....	292
2.12	Characteristics of Hardware, OS and IDEs	293
2.12.1	Selecting the address space for data storage	293
2.12.2	Operating System PxROS	293
2.12.3	Linux Operating System.....	296
2.12.4	Windows Operating System.....	301
3	Hints for Porting to Other Target Platforms.....	312
3.1	Global definition file GLOBAL.H	313
3.2	Selecting the CAN driver	315
3.3	CAN bit rate definition	317
3.4	Target specific settings	318
3.4.1	Hardware properties definition	319
3.4.2	Memory management definition - standard functions.....	319
3.4.3	Definition of target specific functions.....	319
3.5	CPU variable byte order definition.....	321
3.6	Typical configuration of a CANopen slave	321
3.7	Typical configuration of a CANopen master	323
4	Notes on CANopen certification	325
5	Glossary.....	327
6	Revision History CANopen V5.xx	Fehler! Textmarke nicht definiert.
7	Index.....	Fehler! Textmarke nicht definiert.

Index of figures and tables

Figure 1:	Overview of the CANopen concept	10
Figure 2:	Communication model for PDOs	13
Figure 3:	Mapping of object dictionary entries into a PDO	14
Figure 4:	Data transmission of object data via SDO	19
Figure 5:	Structure of an emergency message	22
Figure 6:	“switch mode global” service	23
Figure 7:	“Configure bit timing” service	23
Figure 8:	“Response to Configure bit timing” service	23
Figure 9:	“Activate bit timing” service	23
Figure 10:	“Configure Node ID” service	23
Figure 11:	Response to “Configure Node ID” service	24
Figure 12:	NMT state machine for CANopen devices	25
Figure 13:	Response of the NMT slave to a node guarding remote frame	26
Figure 14:	Response from the NMT slave to a life guarding remote frame	27
Figure 15:	Heartbeat message	27
Figure 16:	Software structure overview	34
Figure 17:	Data exchange between application and object dictionary	42
Figure 18:	Sequence of a typical CANopen application	50
Figure 19:	NMT state machine according to CiA 301 V4.02	55
Figure 20:	Call sequence of the Store Callback-Function for an OD-area	101
Figure 21:	Blinking cycles according to CiA 303-3 (time in ms)	138
Figure 22:	Sequence for NMT events in the NMT callback function	154
Figure 23:	SDO server table	156
Figure 24:	Interfaces for modifying communication parameters of a SDO server	158
Figure 25:	Initiating an SDO download	161
Figure 26:	SDO client table	172
Figure 27:	Interface for changing SDO client parameters	174
Figure 28:	Initiating an SDO download	175
Figure 29:	PDO mapping example of the variables at static PDO mapping	198
Figure 30:	Calling sequence of events for the object callback function during a SDO access	211
Figure 31:	Calling sequence of Events for the object callback function during an access created from the application	211
Figure 32:	Call Sequence of the CCM Functions with PxROS	295
Figure 33:	Structure of CANopen Software under Linux	296
Figure 34:	Call Sequence of the CCM Functions with Linux	298
Figure 35:	CANopen Software structure under Windows	302

Table 1:	Example for mapping parameters for the first TPDO	14
Table 2:	Communication parameter for the first TPDO.....	15
Table 3:	Structure of a COB-ID for PDOs.....	16
Table 4:	Transmission type for TPDOs.....	18
Table 5:	Transmission type for RPDOs	18
Table 6:	SDO transfer types	20
Table 7:	Baud rate table according to CiA 305.....	24
Table 8:	Node state of a CANopen device	26
Table 9:	Heartbeat consumer configuration.....	28
Table 10:	Structure of an object dictionary entry	30
Table 11:	pre-defined master/slave connection set [1]	32
Table 12:	CANopen stack modules	35
Table 13:	CCM layer files	38
Table 14:	part of an object dictionary as example	44
Table 15:	Meaning of instance macros as handle	46
Table 16:	Meaning of Instance macros as pointer.....	46
Table 17:	Guide for selecting the required software modules.....	48
Table 18:	NMT state machine explanation (list of events and commands).....	55
Table 19:	Supported communication objects in various NMT states [4]	56
Table 20:	Parameters of the structure tCmInitParam.....	64
Table 21:	Parameters of the structure tVarParam	68
Table 22:	Description of the argument pointers based on parameter ErrorCode_p ..	72
Table 23:	Parameters of the structure tNmtStateError	73
Table 24:	Parameters of the structure tPdoError	73
Table 25:	Parameters of the structure tSdocParam	80
Table 26:	Parameters of the structure tSdocTransferParam	83
Table 27:	Possible SDO transfer status values in tSdocState	85
Table 28:	Parameter of Structure tSdocNetworkParam.....	88
Table 29:	Parameters of the structure tPdoParam	90
Table 30:	Events for the Lifeguard callback function	95
Table 31:	Assignment of sub-indexes in object 0x1010.....	98
Table 32:	Parameters of the structure tObjCbStoreParam.....	101
Table 33:	Tasks of the callback function CcmCbStoreLoadObject.....	102
Table 34:	Master callback function events.....	107
Table 35:	Parameters of structure tEmcParam	116
Table 36:	Events for callback function CcmCbEmpcEvent.....	119
Table 37:	Parameters of the structure tHbdProdParam.....	122
Table 38:	Event overview and description for heartbeat consumer	124
Table 39:	Return codes for function TgtCavGetAttrib	132
Table 40:	Equivalent function for static PDO mapping	136
Table 41:	Parameter of the structure tPdoStaticParam	137
Table 42:	States of the green LED according to CiA 303-3.....	139
Table 43:	States of the red LED according to CiA 303-3.....	139
Table 44:	Values for parameter State_p of function Ccm303SetRunState.....	141
Table 45:	Values for parameter State_p of function Ccm303SetErrorState	142
Table 46:	Configuration settings for LSS master and slave.....	144
Table 47:	values defined for LSS mode parameter	145
Table 48:	Pre-defined values for the service flags of the LSS master	148
Table 49:	LSS service commands for „service inquire identify“	148
Table 50:	Parameter of Structure tLssCbParam.....	151
Table 51:	Meaning of LSS-Events.....	152
Table 52:	Effects of object properties on the SDO transfer	159
Table 53:	Denial of SDO download initiation at the SDO server.....	160
Table 54:	Denial of SDO Segment download at the SDO server	161
Table 55:	Denial of SDO upload initiation at the SDO server	162

Table 56:	Denial of an SDO segment download at the SDO server	162
Table 57:	Selection of the CRC-calculation algorithm	163
Table 58:	Parameters of structure tSdosInitParam.....	166
Table 59:	Parameters of structure tSdosParam	169
Table 60:	Rejecting the download response by the SDO client.....	174
Table 61:	Rejecting an upload segment by the SDO client	175
Table 62:	Parameters of the structure tSdocInitParam.....	177
Table 63:	NMT events processed by SdocNmtEvent	179
Table 64:	Parameters for the structure tSdocCbFinishParam	182
Table 65:	PDO transmission types and events for sending PDOs	187
Table 66:	Events for calling a PDO callback function (Receipt)	189
Table 67:	Events for calling a PDO callback function (Sending)	189
Table 68:	Partitions of the object dictionary.....	203
Table 69:	Executable instructions to the object dictionary	204
Table 70:	CANopen node states	206
Table 71:	Meaning of the parameter structure tObdCbParam.....	208
Table 72:	Events of the callback function for object access	209
Table 73:	Meaning of the parameter of structure tObdVStringDomain.....	209
Table 74:	Calculating the number of communication objects	212
Table 75:	Parameters of the structure tCobParam	213
Table 76:	Meaning of the communication object types.....	214
Table 77:	Meaning of the NMT commands	218
Table 78:	Meaning of the tNmtmSlaveParam structure parameters.....	222
Table 79:	Meaning of the tNmtmSlaveInfo structure parameters	224
Table 80:	List of available CANopen add-on modules.....	237
Table 81:	Parameters of structure tMPdoParam	238
Table 82:	SDO abort codes.....	248
Table 83:	Emergency error codes according to [4].....	249
Table 84:	Function prefix for the CAN driver	254
Table 85:	Properties for executing process functions.....	276
Table 86:	Setting time monitoring for the PDO module	279
Table 87:	Additional parameters in the structure tCcmlInitParam	294
Table 88:	CCM Thread Events under Linux	301
Table 89:	Thread Events for CANopen under Windows.....	310
Table 90:	Memory type definition for various target systems	314
Table 91:	List of currently available CAN drivers.....	317
Table 92:	Bit rate configuration file overview	318
Table 93:	List of application-specific macros.....	319
Table 94:	List of target-specific functions	320

PREFACE

This manual describes the application layer as well as the supported communication objects of the CANopen stack for programmable CANopen devices. Device profiles are profile-specific and described in a separate manual.

- Section 1 Provides general information on CANopen-related terms and concepts.
- Section 2 Describes the implementation of the CANopen stack protocol by SYS TEC electronic GmbH and gives detailed information about the user functions, their interfaces and data structures.
- Section 3 Provides specific information on how to use and implement the CANopen stack in a user application with regard to the user hardware, the operating system and development environment.

1 CANopen Fundamentals

CANopen is a profile family for industrial communication with distributed automation control devices based on the CAN-bus. It was developed by the manufacturer and users association CiA¹ and has been standardized since 2002 as CENELEC EN 50325-4. CANopen has established itself in a number of areas of industrial communication (e.g. mechanical engineering, drive systems and components, medical devices, building automation, vehicle construction, etc.). The fundamental communications mechanisms are described in so-called communication profiles.

Frameworks complement the communication profile for specific applications. This is how frameworks are defined for safety-compliant data transfer ("CANopen Safety") or for programmable devices (e.g. PLCs). The so-called object dictionary is the central element of every CANopen device and describes the device's functionality.

¹: CAN in Automation e.V. Founded in March 1992, CiA provides technical, product and marketing information with the aim of fostering Controller Area Network's image and providing a path for future developments of the CAN protocol.

1.1 What is CANopen?

CANopen defines the application layer, a communication profile as well as various application profiles.

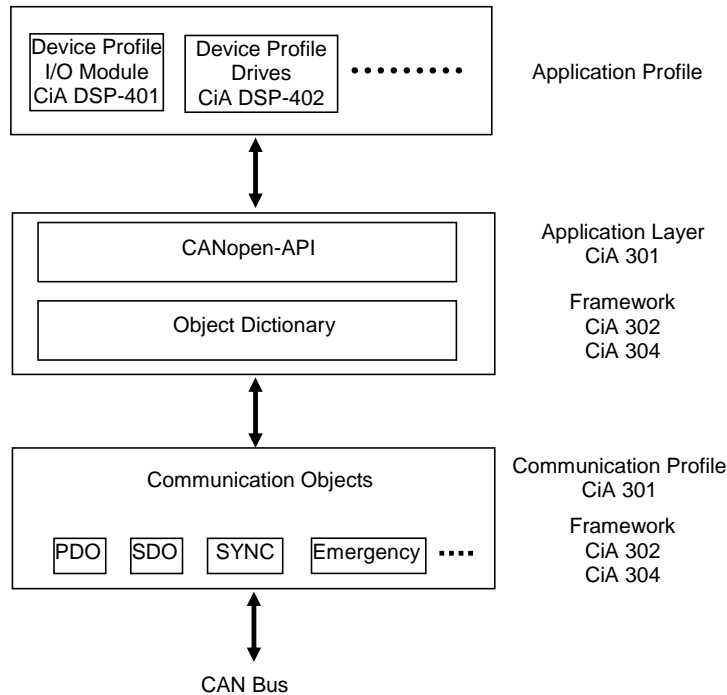


Figure 1: Overview of the CANopen concept

The application layer¹ provides confirmed and unconfirmed services to the application and defines the communication objects. Services are used to, for example, request data from a server.

Communication objects are used for data exchange. Communication objects are available for exchanging process and service data, for process or system time synchronization, for error state supervision as well as for control and monitoring of node states. These objects are defined by their structure, transmission types and their CAN identifier. The specific parameters of a communication object, such as the CAN identifier used for data transmission, the transmission type² of a message, the inhibit time³ or event time⁴ are specified by the communication profile.

The order of and the rules for a data transmission between communication objects are described by protocols (., download, ..).

- 1: The interface to the application (API) is not defined by the application layer and depends on the manufacturer-specific implementation.
- 2: The transmission type defines the properties for initiating a transmission. Available transmission types are cyclic and acyclic as well as synchronous and asynchronous.
- 3: The inhibit time specifies the time that must elapse between two message transmission before a new transmission can be initiated.
- 4: An asynchronous TPDO (transmit PDO) will be sent after the event time has elapsed.

The application layer and the communication objects do not define the interpretation of the transmitted data, however. Interpretation of these data is defined in the application profile respectively. the device profiles. Device profiles are available for different device classes, such as I/O modules (CiA 401), drives (CiA 402) and human-machine interfaces (HMI) (CiA 403). The standardization of device-specific data interpretation allows the building of exchangeable devices.

Each CANopen device features an object dictionary (OD) as the main data structure. The object dictionary serves as the primary data exchange medium between the application and the CAN bus communication. Access to the OD entries is possible from both sides, from the application as well as from the CAN bus via specific messages. These OD entries can be considered as variables or fields from the programmer's point of view.

Each entry in the object dictionary has an index and a sub-index assigned to it. Using this index structure, it is possible to clearly address an OD entry. The CANopen stack provides API functions¹ to define entries in the object dictionary as well as to read or write these entries. With the help of communication objects it is also possible to access the object dictionary over the CAN bus.

Properties have to be defined for each entry in the object dictionary. These properties include the data type (UNSIGNED8, and various attributes such as the access rights (read-only, write-only, , the transmission of the data in a PDO² or supervision of the value range via its limiting values³.

The application layer and the communication profile are thoroughly described by the CiA DS301 specification. Use of CANopen frameworks extensions of this standard is described for specific applications. These frameworks define further rules as well as specific communication objects. For example, the CiA DS301 defines network management objects (node guarding, life guarding). Use of these objects for supervision of CANopen devices is described by the framework.

The following CANopen frameworks are available:

- Framework for programmable CANopen devices (CiA 302)
- Framework for safety-relevant data transmission (CiA 304)

1: Definition of the API functions is manufacturer-specific.
2: Entries can be „mapped“ into a PDO for transmission as process data object.
3: Only such values are written to an entry if they are within the limiting value ranges. All other values will not be accepted.

Summary of advantages using CANopen:

- vendor-independent standards
- open structure
 - real-time communication for process data without protocol overhead
 - modular, scalable structure that can be tailored to the needs of the user within a wide range of networked automation control systems
 - comprehensive functionality for communication and network supervision tasks
 - support of system integrators by configuration and supervision tools
 - profiles oriented on Interbus-S, Profibus and MMS

CANopen provides the following possibilities for auto configuration of CAN networks:

- easy and unified access to all device parameters
- cyclic and event-driven data transfer
- device synchronization especially for multi-device systems

SYS TEC electronic GmbH offers the following products and services to support customers in the design of their CANopen applications:

- Implementation of own CANopen master and slave nodes
- Independent consultancy
- Development of hardware and software
- System integration and certification support
- CAN / CANopen seminars

The engineers of *SYS TEC electronic GmbH* have many years of experience with a variety of CAN applications and participate in the Special Interest Group SiG "Programmable Devices" and "CANopen Safety".

1.2 Communication objects (COB)

Communication objects ¹ (COB) are used for transmission of data. The communication profile defines the parameters of individual communication objects.

Depending on the communication objects, different transmission types and protocols are available. Connection of communication objects over the CAN bus is accomplished via CAN identifiers. The recipient of a communication object must have the same COB identifier (COB-ID, CAN identifier) as the sender of this message. Communication objects for unconfirming protocols (PDO, Emergency) possess one COB identifier (COB-ID, CAN identifier) while communication objects for confirming protocols (SDO) possess two COB identifiers (one identifier each direction).

1.2.1 Process data object (PDO)

Process data objects (PDO) are especially suited for fast transmission of process data. The communication model for PDOs defines one PDO producer and one or multiple PDO consumers.

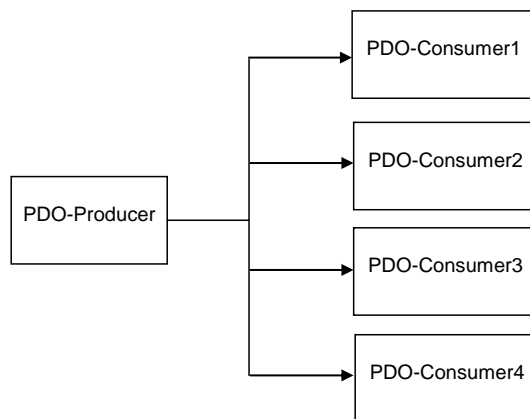


Figure 2: Communication model for PDOs

The reception of a PDO is not acknowledged by the PDO consumer. The PDO producer transmits a PDO, such PDOs are called transmit PDOs (TPDOs). The PDO consumer receives a PDO, consequently such PDOs are called receive PDOs (RPDOs). Successful reception of a PDO is not acknowledged. Multiple PDO consumers may exist for one PDO producer. A PDO producer is assigned to one or multiple PDO consumers with the help of its COB-ID. This is also called PDO linking².

- ¹: CANopen defines different communication objects that are specifically tailored to various tasks and requirements. For example, process data are transmitted without protocol overhead in a single CAN message. Service data objects use additional security mechanisms for supervision of the data transfer between two nodes. The data contents of such an (SDO) object can be transmitted via multiple CAN messages.
- ²: PDO linking can be supported by graphical configuration tools especially for more complex applications requiring many connections between TPDOs and RPDOs.

Transmission of a PDO is triggered by an event. Such events can be the change of a variable that is represented by this PDO, expiration of a time or receipt of a certain message. Process data is transmitted without protocol overhead directly in a single CAN message. The length of a PDO can be between 0 to 8 data bytes.

PDOs are described by their mapping parameters and their communication parameters. The maximum number of TPDOs as well as RPDOs that can be defined is 512. A simple CANopen device typically supports 4 PDOs. The actual number of PDOs is defined by the application or by the device profile for a specific CANopen device.

1.2.1.1 Mapping parameters – What is the structure of a PDO?

A PDO consists of adjacent entries in the object dictionary. The so-called mapping parameters define the connection to these entries. A mapping parameter defines the source of the data via index, sub-index and number of bits. The destination, i.e. the placement within a CAN message, is defined by the order of the mapping parameters in the mapping table as well as the number of bits for each data.

Example:

Index	Sub-index	Object Data	Description
0x1A00	0	4	Number of mapped entries
	1	0x20000310	The entry at index 0x2000, sub-index 3, with a length of 16 bit, is mapped to bytes 0 and 1 within the CAN message.
	2	0x20000108	The entry at index 0x2000, sub-index 1, with a length of 8 bit, is mapped to byte 2 within the CAN message.
	

Table 1: Example for mapping parameters for the first TPDO

A CAN message can contains a maximum of 8 data bytes. This means that when using a PDO, up to 8 object dictionary entries can be transmitted in one PDO.

RPDO

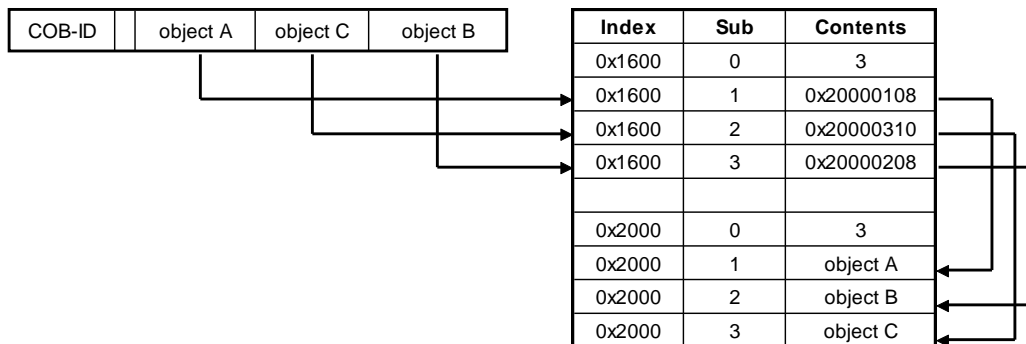


Figure 3: Mapping of object dictionary entries into a PDO

Mapping parameters are entries in the object dictionary (RPDOs: index 0x1600 – 0x17FF, TPDOs: 0x1A00-0x1BFF) and therefore can be read via the CAN bus using service data object (SDO) and, if permitted (if write access is enabled for this entry), be modified as well. The PDO mapping can be done statically. In this case mapping parameters cannot be changed. Depending on the device profile or application specification, it is also possible to change the PDO mapping of a CANopen device at runtime. This is called dynamic mapping¹.

Note:

Before performing a new mapping the user must ensure that PDO is set invalid (by setting bit 31 of COB-ID in its communication parameters) and sub-index 0 of its mapping parameters contains the value 0. If this is not the case, the SDO abort code 0x06010000 (unsupported object access) is returned upon an attempt to remap.

With the help of a SDO download, the new configuration can be stored in the mapping table. The new configuration becomes valid after writing the new value to sub-index 0 in the mapping table, which is unequal to 0.

1.2.1.2 Communication parameter for PDO

The communication parameters define the transmission properties and the COB-IB (CAN identifier) for transmission of a PDO. Configuration of the communication parameters has a direct impact on the frequency of PDO transmissions and hence on the CAN bus load.

Index	Sub-index	Object Data	Description
1800h	0	Number on entries	
	1	COB-ID	CAN identifier for the PDO
	2	Transmission type	transmission type of the PDO
	3	Inhibit time	minimum inhibit time for a TPDO
	4	reserved	reserved
	5	Event time	maximum time between two TPDOs

Table 2: Communication parameter for the first TPDO

PDO communication parameters are entries in the object dictionary (for RPDOs: index 0x1400 – 0x15FF, for TPDOs: index 0x1800-0x19FF) that can be read and, if permitted, changed via the CAN bus with the help of service data object (SDO).

¹: Dynamical mapping requires that the modified mapping parameters are stored on a non-volatile memory on the target device. If this is not possible (no non-volatile memory available) the system configurator must restore the mapping upon network bootup.

1.2.1.3 COB-ID (CAN identifier, sub-index 1)

The COB-ID serves for identification and definition of the PDO's priority upon bus access. Only one sender (producer) is allowed for each individual CAN message. It is, however, possible that multiple receivers (consumers) for this message exist.

Bit	31	30	29	28 – 11	10 - 0
11-bit-ID	0/1	0/1	0	00000000000000000000	11-bit identifier
29-bit-ID	0/1	0/1	1	29-bit identifier	

Table 3: Structure of a COB-ID for PDOs

Bit 30 defines the access rights, bit 30=0 means that a remote transmission request (RTR) for this PDO is permitted. Setting bit 31 to logical 1 the PDO can be deactivated for further processing.

Note:

Since CiA 301 V4.02, a new procedure for changing of the mapping and communication parameters applies.

Before bit 0 to 29 can be changed, a configuration tool needs to set bit 31 of the COB-ID to 1. By doing this, the PDO becomes disabled and it is allowed to change the parameters. The same procedure has to be followed for changing the transmission type (sub-index 2).

The CANopen standard defines COB-IDs (default identifier) for the first 4 PDOs depending on the node number (pre-defined connection set – refer to section 1.8). Communication between slave nodes is not possible when using these default identifiers. CANopen offers the possibility to adjust the CAN identifier for a given communication object. For example, the CAN identifier for a TPDO in a CANopen device can also be assigned to a RPDO in another CANopen device. With this, it is possible to establish direct communication between two slave nodes. This assignment of CAN identifiers for PDOs is also called PDO linking.

1.2.1.4 Transmission type (sub-index 2)

The transmission type of a TPDO defines under which circumstances data are collected (e.g. input values read) and a PDO is transmitted. For RPDOs, the transmission type defines how data received in the PDO is put through to the outputs of the device. Transmission can be initiated event-driven, synchronized or in polling mode.

a) TPDOs

A TPDO can be transmitted cyclic or acyclic. Cyclic transmission takes place after receipt of a cyclic SYNC message¹. In this case, it is unimportant whether input data has changed or not. If the transmission type of a TPDO is set to acyclic, the corresponding TPDO is sent only after a certain event occurred. Such an event can be the reception of a SYNC message, a change of the input data, the expiration of an event timer period² or a remote frame.

b) RPDOs

RPDOs will always be received. However, data contained in the RPDO will only be put through to the corresponding outputs if certain events occur. Such an event can be the reception of a SYNC message or a change of the receipt data compared to the previous RPDO. As an option, the event timer (sub-index 5) can be configured as supervision time for any transmission type. If a PDO is received outside of the period configured with the event time, then the application will be informed (see [CcmCbError](#) section 2.7.1.8).

-
- 1: A SYNC message is a CAN message without data content and is used to synchronize communication objects of other connected nodes. The SYNC producer is responsible for cyclic transmission of the SYNC message.
- 2: An event timer can be used to initiate transmission of a PDO after the event time is expired even if the data within the PDO have not changed. The event time is configured with the help of sub-index 5.
-

Transmission type	Data requisition	Transmit PDO
0	Data (input values) are read upon receipt of a SYNC message.	If the PDO data has changed compared to the previous PDO content then the PDO will be transmitted.
1 – 240	Data is collected and updated upon receipt of the n-th number of SYNC messages and then transmitted on the bus. The transmission type corresponds to the value of n.	
241-251	reserved	
252	Data (input values) are read upon receipt of a SYNC message.	The PDO is transmitted upon request via a remote frame.
253	The application continuously collects and updates the input data.	
254	The application defines the event for data requisition and transmission of a PDO. An event that causes transmission of a PDO can be the expiration of the event timer. The event timer period is configured with sub-index 5. Transmission of a PDO (independent from the event and if the event timer was configured) always starts a new event timer period.	
255	The device profile defines the event for data requisition and transmission of the PDO. An event that causes transmission of a PDO can be the expiration of the event timer. The event timer period is configured with sub-index 5. Transmission of a PDO (independent from the event and if the event timer was configured) always starts a new event timer period.	

Table 4: Transmission type for TPDOs

Transmission type	PDO receipt	Data update
0	The PDO will always be receipt. Analysis and, if required, update of the data occurs upon receipt of the next valid SYNC message.	Data is analyzed upon receipt of a SYNC message. If the data has changed compared to the previous RPDO, then it will be updated on the outputs. Transmission of the SYNC message is acyclic.
1 – 240		Data is analyzed upon receipt of the n-th number of SYNC messages. If the data has changed compared to the previous RPDO, then they will be updated on the outputs. The transmission type corresponds to the value of n. Transmission of the SYNC message is cyclic.
241-251	reserved	
252	reserved	
253		
254	The PDO will always be receipt.	The application defines the event for updating the output data.
255	The PDO will always be receipt.	The device profile defines the event for updating the output data.

Table 5: Transmission type for RPDOs

1.2.1.5 Inhibit time (sub-index 3)

The inhibit time represents the minimum time that must elapse between transmission of two TPDOs. This enables a reduction of the bus load and an increase in data bandwidth.

The inhibit time is stored as UNSIGNED16 value in steps of 100 μ s.

1.2.1.6 Event time (sub-index 5)

a) TPDOs

After the event time has expired, a TPDO is sent, even if the data content of the PDO has not changed compared to the previous transmission. The event timer is restarted after each transmission. Hereby it is unimportant whether the transmission was caused by the expiration of the event time or the change of the PDO data. This allows configuration of periodic PDO transmission. An inhibit time, configured via sub-index 3, will not be considered.

Resetting the event time to zero (zero is the default value) results in deactivation of the event timer. Transmission of the PDO is then only possible if the data content changes. The inhibit time will be considered in this case.

b) RPDO

The event timer (sub-index 5) can be configured as supervision time if the transmission type 254 or 255 is selected. If no PDO is received within the period configured with the event time, then the application will be informed. This feature is not implemented in SYS TEC CANopen stack. However, user can implement it in application level.

1.2.2 Service data object (SDO)

The object dictionary serves as primary data exchange medium between several CANopen devices. All data entries for a CANopen device can be managed within the object dictionary (OD). Each OD entry can be addressed using index and sub-index. CANopen defines so-called service data objects (SDO) that are used to access these entries from another CANopen device.

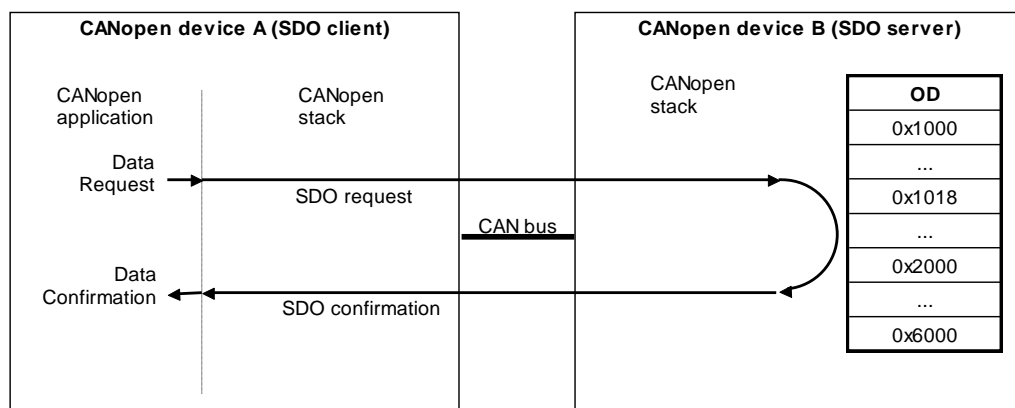


Figure 4: Data transmission of object data via SDO

The communication model used for this data exchange is based on the client-server model. A read or write access is always initiated by a client and is served by a server. Each CANopen device must have an SDO server to access its object dictionary.

SDO transmission requires two different COB-IDs (CAN identifier). The first COB-ID is used to transmit the request from the client to the server. The server sends its response back to the client using the second COB-ID. Different COB-IDs must be used for each direction in order to avoid collisions on the CAN bus. The communication profile defines the COB-IDs that should be used for the default SDO server. Each CANopen device may possess up to 127 SDO servers.

The CANopen standard CiA 301 defines different protocols for transmission of SDOs.

Protocol	Data Length	Description
expedited transfer	1 – 4 bytes	Data is already transmitted when initiating the data transfer. This protocol must be supported by each CANopen device.
segmented transfer	1 - >64 kByte	Only the length of the upcoming data package is transmitted when initiating the data transfer. Data is transmitted in segments of 7 data bytes and one protocol byte each. Each segment is confirmed by a response message.
block transfer	1 - >64 kByte	Only the length of the upcoming data package is transmitted when initiating the data transfer. Data is transmitted in segments of 7 data bytes and one protocol byte each. Up to 127 segments are transmitted within one block. Only complete blocks are confirmed by a response message. Lack of confirmation for each segment increases the data throughput on the bus especially when transmitting larger data packages.

Table 6: SDO transfer types

Reading of OD entries is called 'upload', writing of entries is called 'download'. An ongoing transmission can be terminated by a server or a client with the help of the abort transfer service.

1.2.1 Synchronization object (SYNC)

The synchronization mechanism used in CANopen is based on the producer-consumer scheme. One producer exists in the network that cyclically transmits the SYNC message. The SYNC message contains no data.

The identifier for this SYNC message is specified in object dictionary entry 0x1005. This entry furthermore configures whether the device is SYNC producer or SYNC consumer.

Two other object dictionary entries specify the timing properties during transmission. The time interval between two subsequent SYNC messages is defined in entry *Communication Cycle Time* (0x1006). The time interval in which the TPDOs must be transmitted at the latest after receiving a SYNC message is configured with the *Sync Window* (0x1007) entry.

For each device supporting synchronous PDOs the SYNC message has the following meaning:

TPDOs: update the data to be sent and subsequent transmission of the PDO within the synchronization window

RPDOs: output the data received in the previous PDO during the most recent synchronization interval to the corresponding outputs

1.2.2 Time stamp object (TIME)

CANopen provides a mechanism that allows for synchronization of all network nodes. This service is based on the producer-consumer model. One TIME producer exists in the network that provides the common reference time for all nodes (consumers).

The identifier for the TIME message is defined with object dictionary entry *Time Stamp Object* (0x1012).

1.2.3 Emergency object (EMCY)

CANopen supports the application to indicate error states over the CAN bus. Two error categories can be distinguished:

Communication Error

The network layer can recognize and report the following errors:

- frequent occurrence of errors while transmitting messages
- bus-off state of the CAN controllers¹
- Transmit buffer overflow
- Receive buffer overflow
- Loss of heartbeat or life guarding
- CRC error in SDO block transfer

Application Error

Application errors are errors such as short circuit, under-voltage, exceeding temperature thresholds, code or RAM errors as well as conditions not permitted such as alarms and disturbances.

¹: Each CAN controller has an internal error counter. This error counter is decremented after successful communication. If the error counter exceeds certain error limits it causes the CAN controller to shut off. It then will no longer participate on further communication unless the application resets the CAN controller or its error counter.

The Application and network layer signalize such errors. However, it is the application's task to analyze, process and signalize these errors. CANopen provides the communication object '*Emergency*' to report such errors over the CAN bus.

Identifier	Data							
	0	1	2	3	4	5	6	7
0x080+ Node Number	Emergency Error Code		Error Register	manufacturer-specific information				
	Index 0x1003		0x1001					

Figure 5: Structure of an emergency message

The CiA-301 standard as well as the applicable device profiles for CANopen define specific error codes for transmission of error states. The emergency message can also contain manufacturer-specific data that further describes the error. The transmitted error code indicates the error that occurred. The error register assigns certain categories to groups of errors and indicates if errors still exist within the corresponding category. If the error disappears, the CANopen device will transmit a message with the error code reset (high portion equals zero). At the same time, the data content of the error register that is also transmitted in this message indicates if other errors still exist.

Errors, that are caused by improper access to object dictionary entries or interrupted transmission of SDO services, will be reported by an '*abort SDO transfer service*' message in CANopen.

1.2.4 Layer setting service (LSS)

In the CiA 305 standard CANopen defines layer setting services (LSS) to allow configuration of base parameters (baud rate, node number) for devices that do not provide any means of external mechanical configuration (e.g. via DIP or HEX switches). The LSS master can change the baud rate and node number of a CANopen LSS slave over the CAN bus with the help of layer setting services (LSS). First the LSS master renders all LSS slaves into configuration mode. Then the LSS master transmits the new baud rate using the '*Configure bit timing*' service. The LSS slave now responds with a CAN message that indicates whether this new baud rate is supported by the LSS slave or not. If the LSS slave accepts the new baud rate the LSS master sends the '*Activate bit timing*' service to the LSS slave. This informs the LSS slave to activate the new baud rate after a time called '*switch_delay*'. After successful completion of this cycle the LSS master renders the LSS slave back into operational mode.

The LSS service can also be used to change the node address of an LSS slave. For this, the LSS master renders all LSS slaves into configuration mode again. Then the LSS master transmits the new node address. The LSS slave now responds with a CAN message that indicates whether this new node number is within the supported range of node numbers for this node. Upon switching the LSS slave back into operational mode, a software reset is released. This causes the LSS slave to configure its communication objects based on the new node number (*refer to section 1.8*).

Identifier	DLC	Data							
		0	1	2	3	4	5	6	7
0x7E5	8	0x04	mod	reserved					

Figure 6: "switch mode global" service

mod: new LSS mode
 0 = switch to operational mode
 1 = switch to configuration mode

Identifier	DLC	Data							
		0	1	2	3	4	5	6	7
0x7E5	8	0x13	tab	ind	reserved				

Figure 7: "Configure bit timing" service

tab: indicates the baud rate table to be used
 0 = baud rate table as defined according to CiA 305
 1 ... 127 = reserved
 128 ... 255 = can be defined by the user
ind: index within the baud rate table in which the new baud rate for the CANopen device is stored

Identifier	DLC	Data							
		0	1	2	3	4	5	6	7
0x7E4	8	0x11	err	spec	reserved				

Figure 8: "Response to Configure bit timing" service

err: error code
 0 = operation completed successfully
 1 = baud rate not supported
 2 ... 254 = reserved
 255 = special error code in **spec**
spec: manufacturer-specific error code (only if **err** = 255)

Identifier	DLC	Data							
		0	1	2	3	4	5	6	7
0x7E5	8	0x15	delay	reserved					

Figure 9: "Activate bit timing" service

delay: relative time until activating new baud rate [in ms]

Identifier	DLC	Data							
		0	1	2	3	4	5	6	7
0x7E5	8	0x11	nid	reserved					

Figure 10: "Configure Node ID" service

nid: new node address for the LSS slave (values permitted: 1 to 127)

Identifier	DLC	Data							
		0	1	2	3	4	5	6	7
0x7E4	8	0x13	err	spec	reserved				

Figure 11: Response to "Configure Node ID" service

err: error code
 0 = operation completed successfully
 1 = node address invalid (only values 1 to 127 are permitted)
 2 ... 254 = reserved
 255 = special error code in **spec**
spec: manufacturer-specific error code (only if **err** = 255)

Table Index	Baud Rate	SYSTEC Definition in [cdrv.h]
0	1000 kBit/sec	kBdi1Mbaud
1	800 kBit/sec	kBdi800kBaud
2	500 kBit/sec	kBdi500kBaud
3	250 kBit/sec	kBdi250kBaud
4	125 kBit/sec	kbdi125kBaud
5	100 kBit/sec	kBdi100kBaud
6	50 kBit/sec	kBdi50kBaud
7	20 kBit/sec	kBdi20kBaud
8	10 kBit/sec	kBdi10kBaud

Table 7: Baud rate table according to CiA 305

Note:

The clock speed for various CAN controllers might be different depending on the hardware that is used. Thus any baud rates cannot be selected via LSS service.

The CiA 305 standard also describes further LSS services. Description of these services is not provided in this manual. Please refer to applicable documentation provided by the CiA User's group.

1.3 Network Management (NMT)

Several other network services for supervision of networked nodes are provided in CANopen besides the services for configuration and data exchange. NMT (network management) services require one CANopen device in the network that assumes the tasks of an NMT master. Such tasks include initialization of NMT slave, distribution of identifiers, node supervision and network booting among others.

1.3.1.1 NMT state machine

CANopen defines a state machine that controls the functionality of a device. Transition between the individual states is initiated by internal events or NMT master services. These device states can be connected to application processes.

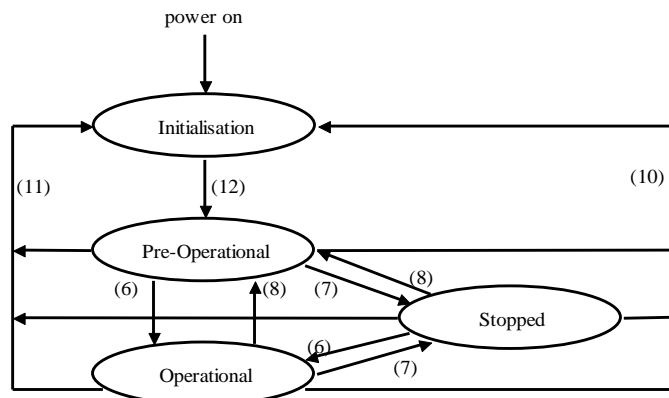


Figure 12: NMT state machine for CANopen devices

In **Initialisation** state, the CANopen data structures of a node are initialized by the application. The CiA 301 standard defines various mandatory OD entries for this task as well as specific communication objects required for that. In the minimum device configuration, the identifier for these communication objects must correspond to the so-called **pre-defined connection set** (refer to section 1.8). The device profiles define further settings for the applicable device class. The pre-defined settings of the identifiers for emergency messages, PDOs and SDOs are calculated based on the node address (Node ID), which can be in the range from 1 to 127, added to a base identifier that determines the function of the individual object.

After **Initialization** is completed the node automatically switches into **Pre-operational** (12) state. The NMT master will be informed about this state change with the BOOTUP message sent by the corresponding node. In this state it is not possible to communicate with the node using PDOs. However, the node can be configured over the CAN bus using SDOs in **Pre-operational** state. NMT services and life guarding are also available in this state.

The application as well as the available resources of the CANopen device determine the amount of configuration via SDO over the CAN bus. For example, if the CANopen device does not provide a non-volatile memory to store mapping and communication parameters for PDOs and these parameters differ from the default values, then these parameters must be transmitted to the node over the network after initialization is completed.

After the configuration of these parameters by the application or the NMT master is completed, the NMT service *start remote node* (6) can be used to render the node from **Pre-operational** state into **Operational** state. This state change also causes the initial transmission of all TPDOs independently of whether an event for it is present. Each subsequent transmission of PDOs then always takes place as a function of an event.

All CANopen devices also support the *stop remote node* (7), *enter pre-operational* (8), *reset node* (10), and *reset communication* (11) services. *reset node* is used to reset the application-specific data and the communication parameter of the node.

The power on values or values stored in non-volatile memory (if previously stored) are used for reset values. The CANopen data structures are loaded with their initial values.

If the NMT service *reset communication* is used to change the state of a node, then communication parameters in the CANopen stack are reset exclusively.

No communication via PDO and SDO is possible if the device is in **Stopped** state. Only NMT services, node guarding, life guarding as well as heartbeat are possible in this state.

1.3.1.2 Node guarding

Node guarding represents a means of node supervision that is initiated by the NMT master. This service is used to request the node's operational state and to determine whether the node is functioning correctly. The NMT master transmits a single Node Guard message to the slave in the form of a remote frame with the CAN identifier 0x700 plus the node address of the NMT slave. As a response to this remote frame, the NMT slave sends a CAN message back containing its current NMT state and a one bit that toggles between two subsequent messages.

Identifier	DLC	Data
		0
0x700 + Node Address	1	Status Byte

Figure 13: Response of the NMT slave to a node guarding remote frame

Status Byte	Node State
0x00	(BOOTUP)
0x04	Stopped
0x05	Operational
0x7F	Pre-operational

Table 8: Node state of a CANopen device

Bit 7 of the status byte always starts with a 0 and changes its value after each transmission. The application is responsible for actively toggling this bit. This ensures that the Node Guard response message from a slave is not just stored in one of the Full-CAN channels. Thus the NMT master will get the confirmation from the NMT slave node that the application is still running.

1.3.1.3 Life guarding

As an alternative to node guarding node supervision can also be performed by life guarding services. In contrast to the node guarding the NMT master cyclically sends a Life Guard message to the slave in the form of a remote frame with the CAN identifier 0x700 plus the node address of the NMT slave. As a response to this remote frame, the NMT slave sends a CAN message back containing its current NMT state and a one bit that toggles between two subsequent messages. The NMT masters application is informed if an answer is missing or in the event of an unexpected status. Furthermore, the slave can detect the loss of the masters. The life guarding is started with the transmission of the first Life Guard message of the masters.

Identifier	DLC	Data
		0
0x700 + Node Address	1	Status Byte

Figure 14: Response from the NMT slave to a life guarding remote frame

Meaning of the status byte corresponds to that of the node guarding message (refer to Table 8).

The life guarding supervision on the NMT slave node is deactivated, if the Life Guard time (object entry 0x100C in the object dictionary) or the Life time factor (object entry 0x100D in the object dictionary) is equal to zero.

1.3.1.4 Heartbeat

Heartbeat is a supervisory service for which no NMT master is necessary. Heartbeat is not based on remote frames, but does work according to the Producer-Consumer model.

1.3.1.5 Heartbeat producer

The heartbeat producer cyclically sends a heartbeat message. The *producer heartbeat time* (16-bit – value in ms), configured at object dictionary index 0x1017, will be used as cycle time between two subsequent heartbeat messages. As COB-ID 0x700 plus node address is used. The first byte of the heartbeat message contains the node status of the heartbeat producer.

Identifier	DLC	Data
		0
0x700 + Node Address	1	Status Byte

Figure 15: Heartbeat message

Meaning of the status byte corresponds to that of the node guarding message (refer to Figure 13).

In contrast to the Node and/or life guarding, bit 7 of the status byte does not change after each transmission. It always contains the value 0. This is also not necessary here, because a Full CAN controller cannot send this message automatically, since this protocol is not based on remote frames. It is the application's task to initiate the transmission of the heartbeat message.

Setting the producer heartbeat time (entry 0x1017 in the object dictionary) to zero disables the heartbeat producer.

1.3.1.6 Heartbeat consumer

The heartbeat consumer analyzes heartbeat messages sent from the producer. In order to monitor the producer, the consumer requires every producer's node number, as well as the consumer heartbeat time.

The information is stored in the object dictionary at entry 0x1016. For every monitored producer, there is a corresponding sub-entry that contains the node number of the producer and the consumer heartbeat time.

Bit	31...24	23...16	15...0
Value	0x00	Node number	Consumer heartbeat time

Table 9: Heartbeat consumer configuration

The consumer is activated when a heartbeat message has been received and a corresponding entry is configured in the OD (value different from 0). If the Heartbeat time configured for a producer expires without receipt of a corresponding heartbeat message, then the consumer reports an event to the application.

The heartbeat consumer is completely deactivated when the consumer heartbeat time is given a value of 0.

1.4 CANopen communication profile

The CiA 301 [4] CANopen communication profile defines the communication parameter for communication objects that must be supported by each CANopen device for this class. Beyond the communication profile supplemental device-specific CANopen frameworks and device profiles are available.

The following CANopen frameworks have been released by the CiA (selection):

- Framework for programmable CANopen devices (CiA 302)
- Framework for safety-relevant data transmission (CiA 304)

The following CANopen device profiles are available:

- Device profile for input/output modules (CiA 401) [7]
- Device profile for drive controls (CiA 402)
- Device profile for display and terminal devices (CiA 403)
- Device profile for sensors and data acquisition modules (CiA 404)
- Device profile for SPS according to IEC 61131-2 (CiA 405)
- Device profile for encoder (CiA 406)
- Device profile for proportional valves (CiA 408)

CAN identifier of a COB, inhibit times and transmission type of a PDO, amongst others, are considered communication parameters. The communication parameters are part of the object dictionary and they can be read from and, if the applicable access rights are granted, be written to by the user application. Some parameters are explained in *section 1.2*, while information on other parameters can be found in the previously discussed CANopen frameworks and device profiles.

1.5 Transmission protocols

Transmission of communication objects is defined by transmission protocols. These protocols are also described in the CiA 301 CANopen communication profile and are not a topic of this manual.

It should be noted, however, that the range of the realizable protocols could be limited. This saves resources for code and data. *Section 2.11* describes how this resource reduction can be achieved.

1.6 Object dictionary (OD)

The object dictionary (OD) is the connecting element between the application and communication on the CAN bus, enabling data exchange from the application over the CAN network. CANopen defines services and communication objects for accessing the object dictionary. Each entry is addressed via index and sub-index. The properties of an OD entry are defined by a type (UINT8, UIN16, REAL32, visible string, and attributes (read-only, write-only, const, read-write, mappable).

The maximum number of OD index entries is 65,536, between 0 and 255 sub-index entries are possible for each (main) index. Index entries are pre-defined by the applicable communication profile or device profile, respectively. Type and attributes for available sub-index entries within a main index may vary.

Index	Sub-index	Type	Attribute
0x2000	0	UINT8	const
	1	UINT32	read-write
	2
	3		

Table 10: Structure of an object dictionary entry

Default values can be assigned to individual entries. The value of an entry can be changed with the help of SDO communication if the attribute assigned to the entry allows such access (read-write and write-only; not possible for read-only and const). The value can also be changed by the application itself if the attributes for the entry are read-write, write-only and read-only (not possible for const).

The OD is further divided in sections. The section with index 0x1000 – 0x1FFF is used for definition of parameters for the communication objects and the storage of common information, such as manufacturer name, device type, serial number etc. Entries from index 0x2000 to 0x5FFF are reserved for storing manufacturer-specific values. Device-specific entries, as defined by the device profile or frameworks, follow at index 0x6000 and higher.

CiA 301 defines several mandatory entries that each CANopen device must always possess. These entries are marked as mandatory. These mandatory entries are supplemented by entries defined in the corresponding device profile.

The creation of an object dictionary is the subject of an additional manual (L-number L-1024) provided by SYS TEC. Creation of an object dictionary from an EDS (electronic data sheet) is supported by the OD-Builder¹ (refer to manual L-1022).

¹: OD-Builder is a product developed by SYS TEC electronic GmbH.

1.7 Error handling and reporting

Various mechanisms are provided in CANopen to report error events:

- **Emergency object:** This is a high-priority, 8-byte message that contains the error information. *Refer to section 1.2.3 for detailed description.*
- **Error register:** This is a 1-byte object dictionary entry at index 0x1001. This entry is provided to report the presence of an error and its type.
- **Pre-defined error field:** This is an error list, which is stored in the object dictionary at index 0x1003. This list contains the emergency error code as well as device-specific information. The structure of this list shows the most recent error at sub-index 1.

1.8 Telegram table (pre-defined connection set)

CANopen defines default COB-IDs (CAN identifier) for simple network configuration with one master node and up to 127 slave nodes. These default COB-IDs depend on the service and the node number of the corresponding slave device. A function code has been defined for each service. The resulting COB-ID is based on the function code and the node number¹.

COB Identifier (CAN Identifier)										
10	9	8	7	6	5	4	3	2	1	0
Function Code				Node Number						

Object	Function Code	Node Number	COB-ID	object dictionary Index
Broadcast messages				
NMT	0000 _b	-	0	-
SYNC	0001 _b	-	0x80	0x1005, 0x1006, 0x1007
TIME STAMP	0010 _b	-	0x100	0x1012, 0x1013
Point-to-point messages				
Emergency	0001 _b	1-127	0x81-0xFF	0x1014, 0x1015
TPDO1	0011 _b	1-127	0x181-0x1FF	0x1800
RPDO1	0100 _b	1-127	0x201-0x27F	0x1400
TPDO2	0101 _b	1-127	0x281-0x2FF	0x1801
RPDO2	0110 _b	1-127	0x301-0x37F	0x1401
TPDO3	0111 _b	1-127	0x381-0x3FF	0x1802
RPDO3	1000 _b	1-127	0x401-0x47F	0x1402
TPDO4	1001 _b	1-127	0x481-0x4FF	0x1803
RPDO4	1010 _b	1-127	0x501-0x57F	0x1403
SDO (tx)	1011 _b	1-127	0x581-0x5FF	0x1200
SDO (rx)	1100 _b	1-127	0x601-0x67F	0x1200
NMT Error Control	1110 _b	1-127	0x701-0x77F	0x1016, 0x1017

Table 11: pre-defined master/slave connection set [1]

¹ The node number can be assigned locally or with the help of LSS services over the CAN bus.

2 CANopen User Layer

The following section describes the data structures and API functions of the *SYS TEC electronic GmbH* specific implementation of the CANopen standard CiA 301. Support for additional CANopen standards is also implemented or prepared. In addition hardware and compiler specific characteristics are taken into consideration as well. The API offers interfaces that can be used for expansion of device specific properties. The experience of *SYS TEC* engineers in integrating or porting the CANopen stack in various customer applications has contributed to an expansion of the standard as well. Therefore any deviations from the CANopen standard are especially identified as such. Design, creation and configuration of an object dictionary is described in a separate manual (*refer to L-1024*).

2.1 Software structure

Before the individual API functions can be explained, a description of the software structure and the file structure is necessary. This provides a foundation for finding your way in later implementation. As a rule, the CANopen stack has a divided structure for application specific and hardware specific modules.

The CANopen stack is divided up into individual modules. With the definition of modules, the CANopen stack's parameters (function parameters, data parameters) were structured so as to be scalable. A portion of the modules are to be considered as core modules and are a mandatory component in the CANopen stack. Other modules are not required for setting tasks. This refers mostly to CANopen functions, which according to the CANopen standard can be implemented optionally or as an alternative to other functions.

In order to leave out individual modules without complications, there can be no lateral function call to another module within the modularized software layer, rather only to modules positioned above or below (as a callback function)¹⁷.

The application specific layer "CANopen controlling module" (CCM) controls the interaction of the individual modules. The CCM layer is not absolutely necessary for implementation in the application. However it provides a convenient interface for use of multiple CANopen instances and encapsulates sequential function calls of multiple API functions (i.e. initialization, definition of PDOs) in functions.

The hardware specific layer encapsulates the special properties of a CAN controller or microcontroller. Porting to new hardware is simplified thereby and can be reduced to an exchange of the transceiver for the CAN controller and the microcontroller specific initialization.

¹⁷: With this it is possible to not include certain modules or services when creating a CANopen application without getting error messages from the linker about unreferenced functions.

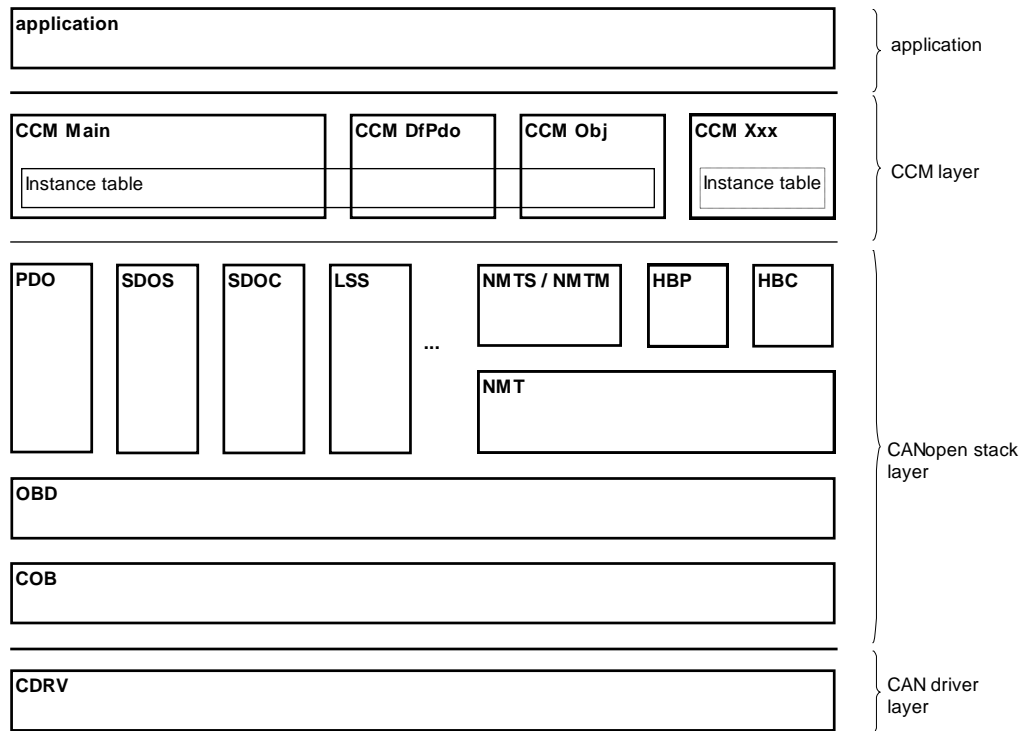


Figure 16: Software structure overview

All software layers are instanceable. This means that multiple CAN networks can be managed within one CANopen device. Combining the several CAN networks has to be done within the application.

2.1.1 CANopen stack

The CANopen stack is portable; this means it is implemented independent from any hardware or application specific environment.

Following table lists all available modules of the SYS TEC CANopen stack, which implements the individual services. All these modules are combined in CCM layer. To add a CANopen service into an own CANopen application user has to add the according module to its project and to set the according bit in constant CCM_MODULE_INTEGRATION of file COPCFG.H.

Module	Description
COB	The COB layer provides services for transmission of communication objects and therefore serves as a base layer that is required in any of the configuration variants.
OBD	The OBD module provides the global data structure for all CANopen instances. All data structures, that are configurable by the user, are created in this module. This includes the object dictionary as well as tables for managing PDOs and SDO server and clients.
NMT	This module creates the NMT state machine and calls the callback function for the NMT state change in the CCM module.
NMTS	This module provides services for node guarding, life guarding and Boot-up as NMT slave. It is not possible to use both NMTS and NMTM at the same time within one CANopen instance.
NMTM	This module provides services for node guarding, life guarding and Boot-up as NMT master. It is not possible to use both NMTM and NMTS at the same time within one CANopen instance.
HBP	This module provides services for a heartbeat producer. It is possible to have a heartbeat producer and a consumer both existing at the same time in one CANopen instance. It is not possible to activate both heartbeat and life guarding at the same time for the given node.
HBC	This module provides services for a heartbeat consumer. It is possible to have a heartbeat producer and a consumer both existing at the same time in one CANopen instance. It is not possible to activate both heartbeat and life guarding at the same time for the given node.
PDO	This module provides services to define and transmit PDOs. In addition, services for Sync Producer and Consumer are generated here as well.
PDOSTC	This module provides the same services as the PDO module but implements a static PDO mapping.
SDOSCOMM	This module provides services to manage SDO servers and service data object (SDO) as well as the protocols for transmission of service data object as server. The supported protocols (expedited, segmented, block) are configurable.
SDOC	This module provides services to manage SDO clients and service data object (SDO) as well as the protocols for transmission of service data object as clients. The supported protocols (expedited, segmented, block) are configurable.
LSSSLV	This module provides services for configuration of bit timing and module ID for a LSS slave.
LSSMST	This module provides services for configuration of bit timing and module ID for a LSS master.
EMCC	This module provides services for an emergency consumer. It is possible to have an emergency producer and consumer both existing at the same time in one CANopen instance.
EMCP	This module provides services for an emergency producer. It is possible to have an emergency producer and consumer both existing at the same time in one CANopen instance.
HPT	This module provides services for a High Precision Time Stamp producer and/or consumer.
TSO	This module provides services for a Time Stamp Object producer and/or consumer.

Table 12: CANopen stack modules

2.1.2 Hardware-specific layer

The CDRV modules make a single interface available to the CANopen stack for various CAN controllers. The special properties and "peculiarities" of the CAN controllers are thus taken into account in the CDRV driver. Porting to a new hardware platform is enabled by creating or adapting the CDRV driver.

The CDRV drivers are instanceable. This solution becomes interesting for targets with multiple CAN controllers. There multiple CANopen interfaces can be created in order to serve multiple CANopen networks from a single application. The implementation of multi-channel CAN cards on the PC (such as pcNetCAN, PCI-CAN or USB-CANmodul) is then possible.

When creating/configuring the CANopen stack, the following cases should be taken into consideration:

- The target supports various CAN controllers (e.g. microcontroller C167CR with integrated CAN controller and an external CAN controller SJA1000). A hardware driver is required for each CAN controller. One instance exists for each hardware driver.
- The target supports more than one CAN controller (e.g. C167CS with two integrated CAN controllers). However, a hardware driver with N instances is required for the CAN controller.

Section 2.11 describes the settings for the selection and configuration of the hardware drivers. For additional information on the CDRV module refer to L-1023 "CAN Driver Software Manual".

2.1.3 Application-specific layer

The application specific layer "CANopen Controlling Module" (CCM module) controls the interaction of the individual modules. The CCM layer is not absolutely necessary for implementation in the application. However, it provides a convenient interface for use of multiple CANopen instances and encapsulates sequential function calls of multiple API functions (i.e. initialization, definition of PDOs) in functions.

The CCM layer contains a series of small function modules. When the application is created, the user can attach suitable modules or use them as models for their own expansions to the CCM layer. These expansions may affect the reaction to certain events, which could occur during a CANopen process. In any case, it is not necessary that the entire set of modules be attached to an application. ¹⁸

Module	Description	Functions
CcmMain.c, CcmWin.c, CcmPxRos.c, CcmLinux.c, CcmWinCe.c	This module contains the global initializing and process functions for CANopen as well as the response to important events (state change of the NMT state machine, transmission errors, state). When using an operating system the according file has to be used instead of CcmMain.c.	- <i>CcmInitCANopen()</i> - <i>CcmShutDownCANopen()</i> - <i>CcmDefineVarTab()</i> - <i>CcmConnectToNet()</i> - <i>CcmProcess()</i> - <i>CcmCbNmtEvent()</i> - <i>CcmCbError()</i>
CcmObj.c	This module contains functions for accessing the own object dictionary.	- <i>CcmWriteObject()</i> - <i>CcmReadObject()</i>

¹⁸: The way of not using software modules that are not required for a specific applications is partially supported by the linkers. This means that a module can be included within an IDE project but will not be included in the linking process when no function call to this module is performed.

Module	Description	Functions
CcmDfPdo.c	This module contains a function for configuring the PDOs via a predefined table in application when using dynamic PDO mapping.	- <i>CcmDefinePdoTab()</i>
CcmSnPdo.c	This module contains helper functions for signalling to send a TPDO when using dynamic PDO mapping.	- <i>CcmSignalCheckVar()</i> - <i>CcmSignalVar()</i> - <i>CcmSendPdo()</i>
CcmStPdo.c	This module contains helper function for configuring/signalling the PDOs via a predefined table in application when using static PDO mapping.	- <i>CcmDefineStaticPdoTab()</i> - <i>CcmSignalStaticPdo()</i> - <i>CcmSendPdo()</i>
CcmStore.c, CcmStore2.c	This module defines functions for storing object data from the object dictionary in the non-volatile memory. CcmStore2.c has to be used instead of CcmStore.c for using POSIX functions to store the OD data into files.	- <i>CcmInitStore()</i> - <i>CcmStoreCheckArchivState()</i> - <i>CcmCbStore()</i> - <i>CcmCbRestore()</i> - <i>CcmCbStoreLoadObject()</i>
CcmSync.c	This module defines functions for the SYNC consumer. It supports the SYNC configuration.	- <i>CcmInitSyncConsumer()</i> - <i>CcmConfigSyncConsumer()</i> - <i>CcmConfigSyncProducer()</i> - <i>CcmCbSyncReceived()</i>
CcmEmcc.c	This module defines functions for the emergency consumer. It supports the creation of a list containing CANopen devices to be monitored.	- <i>CcmInitEmcc()</i> - <i>CcmEmccDefineProducerTab()</i> - <i>CcmCbEmccEvent()</i>
CcmEmcp.c	This module supports configuration of the emergency producer. It provides a function to erase the pre-defined error field.	- <i>CcmConfigEmcp()</i> - <i>CcmSendEmergency()</i> - <i>CcmClearPreDefinedErrorField()</i> - <i>CcmCbEmcpEvent()</i>
CcmNmtm.c	This module contains functions for NMT master services. It also includes a default callback functions for handling NMT master events.	- <i>CcmInitNmtm()</i> - <i>CcmDefineSlaveTab()</i> - <i>CcmSendNmtCommand()</i> - <i>CcmTriggerNodeGuard()</i> - <i>CcmConfigLgm()</i> - <i>CcmCbNmtmEvent()</i>
CcmBoot.c	This module contains a function to send NMT Start Remote Node service using a NMT slave. This is needful when no NMT master is available on CANopen network.	- <i>CcmBootNetwork()</i>
CcmHbc.c	This module defines functions for the heartbeat consumer. It supports the creation of a list containing CANopen devices to be monitored.	- <i>CcmInitHbc()</i> - <i>CcmHbcDefineProducerTab()</i> - <i>CcmCbHbcEvent()</i>
CcmHbp.c	This module supports configuration of the heartbeat producer.	- <i>CcmConfigHbp()</i>
Ccm303.c	This module defines functions needed for indicating the internal states of the CANopen device. Two LEDs display the state information according to the CiA-303.3 standard.	- <i>Ccm303InitIndicators()</i> - <i>Ccm303ProcessIndicators()</i> - <i>Ccm303SetRunState()</i> - <i>Ccm303SetErrorState()</i>

Module	Description	Functions
CcmLss.c	This module provides functions for implementing the LSS master service. The module also contains a default callback function of the LSS slave service.	<ul style="list-style-type: none"> - CcmLssmSwitchMode() - CcmLssmConfigureSlave() - CcmLssmInquireIdentity() - CcmLssmIdentifySlave() - CcmCbLssmEvent() - CcmCbLsssEvent()
CcmLgs.c	This module provides functions for life guarding service. The module also contains a default callback function for handling life guarding events.	<ul style="list-style-type: none"> - CcmInitLgs() - CcmConfigLgs() - CcmCbLgsEvent()
CcmTso.c	This module contains helper functions for implementing the Time Stamp Object service as producer or consumer. The application has to control to send the Time Stamp Object as producer.	<ul style="list-style-type: none"> - CcmTsoConfigConsumer() - CcmTsoConfigProducer() - CcmTsoSend()
CcmSdoc.c	This module contains helper functions for implementing SDO client services.	<ul style="list-style-type: none"> - CcmSdocDefineClientTab() - CcmSdocStartTransfer() - CcmSdocAbort() - CcmSdocGetState()

Table 13: CCM layer files

This list gives the names of a few important files in the CCM layer. The CCM layer contents is expanded constantly and can therefore not be considered to be complete. The description of functions, parameters and implementation can be found in the applicable CCM module.

2.2 Directory structure

Where to find which files?

Folder	Contents
\ccm	Files of the CCM layer.
\copstack	Files of the CANopen stack.
\cdrv	Files of the CAN driver layer.
\examples	Files with sample applications for microcontroller projects and products with operation systems. Here, the C-files are stored together with the main-function. There are additional Header files, i.e. definitions for an easier handling of the CANopen stack (e.g. bditabdf.h and appmco.h).
\include	This folder contains all interface files for CANopen. The files <code>global.h</code> , <code>cop.h</code> must be included in the application.
\objdicts ...	This folder contains predefined object dictionaries for different device profiles. Each object dictionary consist of 3 files that belong together; <code>objdict.c</code> , <code>objdict.h</code> and <code>obdcfg.h</code> . These files can be automatically created with the help of the ODBuilder tool ¹⁹ . The selection of the object dictionary occurs by defining the applicable include path within the project settings. In addition the following subfolders contain the corresponding EDS file and the project file for the ODBuilder.
\domain_string	Object dictionary with domain- and string-objects in the manufacturer-specific area
\ds401_3p	Object dictionary for CiA 401 with 3 RPDOs and 3 TPDOs, NMT slave
\ds401_4pstc	Object dictionary for CiA 401 static PDO mapping, NMT slave
\ds401_7p	Object dictionary for CiA 401 with 7 RPDOs and 7 TPDOs, NMT slave
\o401p3m	Object dictionary for CiA 401 with 3 RPDOs and 3 TPDOs, NMT master
\o401p7m	Object dictionary for CiA 401 with 7 RPDOs and 7 TPDOs
\ds401_4pstc	Object dictionary for CiA 401 static PDO mapping, NMT slave
\o401p3s_hpt	Object dictionary for DSP-401 with 3 RPDOs and 3 TPDOs, NMT-Slave und High Precision Time Stamp
\target ...	This folder contains the project folders for various example applications. One configuration file (<code>copcfg.h</code>) is provided for each project. This file defines the supported hardware, the supported properties and protocols.
\easykit-xc164cm	Demo projects for Infineon Easy Kit XC164CM.
\fujitsu_dev-kit16	Demo projects for Fujitsu Development Kit 16.
\phycore-lpc2294	Demo projects for Phytex phyCORE LPC2294.
\... \x86	Demo projects for PC e.g. using Windows OS.
\projects ...	This folder is obsolete but still exists for compatibility reason.

The included files have been linked to the C files without any path indication. In order to guarantee an error free compilation, the path must be defined to point to the include folder and the object dictionary for the compiler or for the IDE project.

¹⁹: The ODBuilder tool supports the generation of an object dictionary based on an EDS file. The user can also define entries in the OD. The ODBuilder creates a new EDS file as well as the C and header files necessary to create the CANopen data structures.

2.3 Data structures

In the following section there are explanations for the data structures. There are data structures that are used for data exchange between the application and CANopen. Other data structures are used for management and control of processing cycles, functions or protocols within a module, and are only mentioned to provide a complete listing.

The following data structures are used as application interfaces:

- Each CANopen instance²⁰ has its own **object dictionary** (OD). The object dictionary is the coupling element between the application and the communication layer and contains all CANopen device data. Entries in the object dictionary are addressed over index and sub-index. Entries can be read or written over the CAN bus with the help of service data object (SDO, *refer to section 1.2.2*) or through the application with the help of API functions (*refer to sections 2.7.4 and 2.8.5*). With the help of the OBD module's API functions, the address and size of an entry can be determined, whereby access to the object entry data is possible via pointers (*refer to section 2.8.5*). OD entries can also be linked with application fields or variables. This is advantageous in that access to data is possible without using one of the CANopen stacks or a pointer's API functions. Transmission per SDO or access with the help of API functions is not limited thereby. Due to versatility in application and the alterability of these entries they are defined as Var entry (variable object entry).

As mentioned above, these Var entries can be embedded in PDOs; under the condition that mapping of the entries with the attribute *kObdAccPdo* is allowed.

In order to register a fast and simple modification of a variable with the application, a variable callback function that includes an argument pointer can be provided when defining Var-Entries. Modifying an entry about the CAN bus via a PDO results in the call of the respective PDO callback function, whereby the argument pointer is given as the parameter.

The object dictionary is organized as a table. Each table entry corresponds to an index. This index table is located in the ROM. Within an index there are additional tables with an entry for each sub-index. The sub-index table can be stored in either the ROM or the RAM. The design of the table has been optimized for access speed and memory space requirements. Creation of an object dictionary is supported with the help of macros. It can be created manually or by help of the ODBuilder tool.

An entry for a sub-index contains the type of the object dictionary, right of access, start value, range values and the data pointer. In the case of a static object dictionary, the management structure for the object dictionary is created during compilation.

Modification of the table during runtime is not possible. Therefore any later use of an entry must be known about ahead of time. In the case of a dynamic OD, the management structures of the object dictionary are created during runtime.

²⁰: The CANopen stack and the hardware drivers are instanceable. This means that the functional contents of CANopen can be utilized in several data instances. This makes it possible to use various independent CANopen interfaces on the same target (e.g. device with more than one CAN controller).

The application's variables and fields, which are to be transmitted with the help of PDOs or which were declared as DOMAIN or strings, have to be registered in the object dictionary Var entries²¹. The CCM layer makes the function **CcmDefineVarTab** available for this purpose, which automates this procedure with the help of tables.

- **Structures** are used for transferring complex parameters to functions. The structures are explained as function parameters. Prior to a function call, a structure of this kind must be initialized.

Structures and tables for the management of internal cycles and settings:

- For the management of PDOs, SDO server, SDO client, ..., internal tables are used. The size of the tables (i.e. number of entries) is based on the defined number of PDOs, SDO servers etc. (*refer to section 2.11*). The tables are created with the compiler when compiling the object dictionary. In order to conserve memory resources and processing time, individual entries from the tables are connect directly with the entries of the object dictionary. The tables are initialized with the initialization function of the relevant module.
- Each module contains a global instance table. The instance table contains all of the variables for module. The variables are used to store processing states and parameters within a module. Except for in the case of the CCM module, an instance table is only valid within a module and is therefore declared to be "static". Creation and modification of entries for a table is supported by macros (*refer to section 2.5*).

²¹: When creating the object dictionary the data structures for for managing the variables are created but NOT the memory (this means the variable or the field) for storing the data.

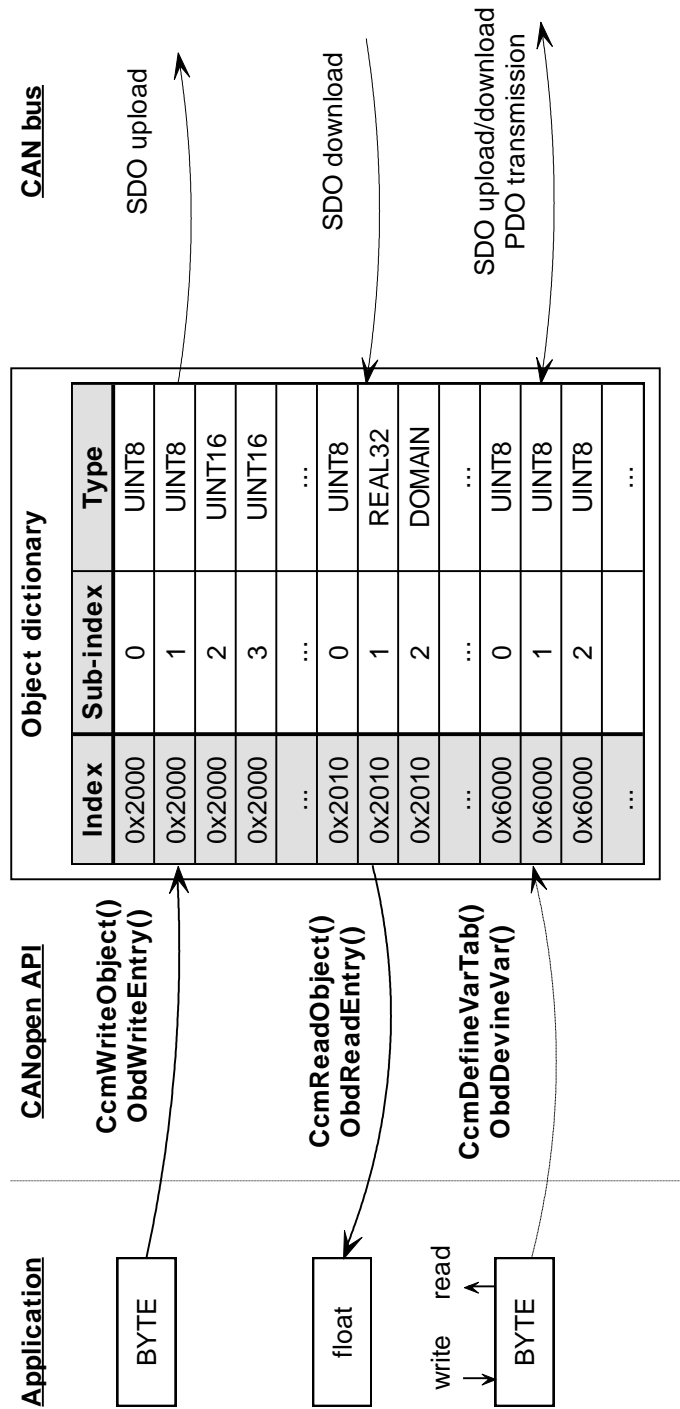


Figure 17: Data exchange between application and object dictionary

2.4 Object dictionary

The object dictionary is defined in three files **objdict.c**, **objdict.h** and **obdcfg.h**. An exact description for the object dictionary creation is given in manual *L-1024*.

CANopen software comes in three standard variants. These variants are listed in the following sections. In the listing of objects, abbreviations are used for the object type, the data type and for the attributes. These abbreviations have the following meanings:

Object Types:

var..... Object contains a value that can be accessed per SDO or from the application (variable).

Data Types:

u8 Unsigned 8-bit

u16 Unsigned 16-bit

u32 Unsigned 32-bit

i8..... Integer 8-bit

i16..... Integer 16-bit

i32..... Integer 32-bit

vstr..... Visible String

Attribute:

ro read only; object can be read per SDO and read or written from the application.

rw..... read write; object can be read or written per SDO or from the application.

wo..... write only; only a write to the object is possible per SDO or from the application.

const..... constant; object can only be read and not written per SDO or from the application.

mapp..... object can be mapped to a PDO

store..... object can be saved in non-volatile memory (*refer to section 2.7.6*)

2.4.1 Example object dictionary

There are several Object Dictionaries available for standard I/O devices. The OD **ds401_3p** contains 3 TPDOs and 3 RPDOs; the OD **ds401_7p** contains 7 TPDOs and 7 RPDOs. Otherwise the two Object Dictionaries are the same in terms of all other objects. The OD **o401p3m** has 3 TPDOs and 3 RPDOs, but as a CANopen Master it does not contain the objects 0x100C and 0x100D. Instead it contains the supplemental objects 0x1016 (for the heartbeat consumer) and 0x1280 (for the first SDO client). **O401p7m** resembles **o401p3m**, except that it has 7 TPDOs and 7 RPDOs. The CANopen Kits have two ODs available to them, **o401p2ks** (for the Slave) and **o401p2km** (for the Master). Both of these contain only 2 TPDOs and 2 RPDOs and the Master is not equipped with a heartbeat consumer (Object 0x1016 is absent).

Index	Sub-index	Name	Object Type	Data Type	Attribute	Default Value
0x1000		device type	var	u32	ro	0x000F-0191
0x1001		error register	var	u8	ro	0
0x1003		pre-defined error field	array			
	0	number of errors	var	u8	ro, rw; write 0 to erease	0
	1...4	standard error field	var	u32	ro	0
0x1005		COB-ID SYNC	var	u32	rw, store	0x080
0x1006		communication cycle period	var	u32	rw, store	0
0x1007		synchronous window length	var	u32	rw, store	0
0x1008		manufacturer device name	var	vstr	const	"CANopen Slave"
0x1009		manufacturer hardware version	var	vstr	const	"V1.00"
0x100A		manufacturer software version	var	vstr	const	"V5.xx"
0x100C		guard time	var	u16	rw, store	0

Table 14: part of an object dictionary as example

2.5 Instanceability of the CANopen layer

The CANopen stack, the CCM module and the hardware drivers are instanceable. This means that the function contents of CANopen can be applied to multiple data instances. This allows for support of multiple independent CANopen interfaces on one target.

To generate instances, all global and static variables are stored in so called instance tables. Each table entry corresponds exactly to a CANopen instance. An entry is described by a structure. When called, the functions receive a reference to the instance to be processed in the form of an instance pointer or an instance handle.

The number of instances and thereby the number of entries in an instance table are defined as constants during compilation. These constants are called `COP_MAX_INSTANCES` for the CANopen and are defined in the file `copcfg.h`. There is a separate define called `CDRV_MAX_INSTANCES` for instancing the CAN drivers, which is also defined in the file (*refer to section 2.11.1*). Access to the structure elements of an instance occurs exclusively via macros.

When defining multiple instances, if a function call occurs, a reference to the instance to be processed is always given as a parameter in the form of an address to an instance table in COPstack modules (*refer to section 2.5.2*) or instance handle in CCM layer (*refer to section 2.5.1*). If only one instance was defined, then this parameter is left out. In the description of the API functions, this parameter will always be listed. The definition of the instance parameter is given with the help of macros. These macros are deleted by the compiler's pre-processor depending on the defined number of instances.

Example:

If only one instance is used, then the following instance parameter should be removed.

```
CcmConnectToNet ();
```

For multiple instances the instance parameter must be given.

```
CcmConnectToNet (HandleInstance0);
```

In the file `instdef.h` macros are defined for the declaration and transmission of instance parameters to functions and for access to entries in an instance table. Use of these macros supports function writes, which are independent from the number of instances. As a rule, the number of instances (CANopen interfaces) is defined by the application.

2.5.1 Using the instance handle

An instance handle is used as a reference to the current instance if a CCM layer function is called or if one of the application's callback functions is called.

If multiple instances are used in a CANopen application, then the instance macros have the following contents:

The macro ...	corresponds to in the C Source
For declaration of parameters in a function's parameter list:	
CCM_DECL_INSTANCE_HDL	<i>tCopInstanceHdl InstanceHandle</i>
CCM_DECL_INSTANCE_HDL_	<i>tCopInstanceHdl InstanceHandle,</i>
CCM_DECL_PTR_INSTANCE_HDL	<i>tCopInstanceHdl MEM* pInstanceHandle</i>
CCM_DECL_PTR_INSTANCE_HDL_	<i>tCopInstanceHdl MEM* pInstanceHandle,</i>
For handing over parameters to the function to be called:	
CCM_INSTANCE_HDL	<i>InstanceHandle</i>
CCM_INSTANCE_HDL_	<i>InstanceHandle,</i>

Table 15: Meaning of instance macros as handle

If only one instance is used then the instance macros have no content.

In an application always using more instances than one you do not have to use the macros but you can directly use corresponding C source described in table above.

2.5.2 Using instance pointers

An instance pointer is used as a reference to the current instance if a function from a deeper layer is called (i.e. **SdosProcess** function call through a function from the CCM module).

If multiple instances are used in a CANopen application, then the instance macros have the following contents:

The macro ...	corresponds to in the C Source
For declaration of parameters in a function's parameter list:	
MCO_DECL_INSTANCE_PTR	<i>void MEM* pInstance</i>
MCO_DECL_INSTANCE_PTR_	<i>void MEM* pInstance,</i>
MCO_DECL_PTR_INSTANCE_PTR	<i>void MEM* MEM* pInstancePtr</i>
MCO_DECL_PTR_INSTANCE_PTR_	<i>void MEM* MEM* pInstancePtr,</i>
For handing over parameters to the module's own function:	
MCO_INSTANCE_PTR	<i>pInstance</i>
MCO_INSTANCE_PTR_	<i>pInstance,</i>
MCO_PTR_INSTANCE_PTR	<i>pInstancePtr</i>
MCO_PTR_INSTANCE_PTR_	<i>pInstancePtr,</i>
For handing over parameters to functions not inside the module:	
MCO_INSTANCE_PARAM(<i>par</i>)	<i>par</i>
MCO_INSTANCE_PARAM_(<i>par</i>)	<i>par,</i>

Table 16: Meaning of Instance macros as pointer

If only one instance is used then the instance macros have no content.

2.6 Hints for creating an application

When using the CANopen layer, it is important to know which functions must be executed in which operating state. This is crucial in order to attain the desired functionality. Explanations of internal mechanics and cycles aid in development of an understanding of the chosen solution or its limitations. Furthermore, explanations are given as to which tasks must be performed by the user in order to achieve the desired function.

To ensure the correct function of the CANopen protocol, a specific sequence must be adhered to when executing the functions. Otherwise it is possible that data structures won't be present or won't be initialized, whereby a function call will result in an error or undefined behavior.²³

The sequence for execution of the various functions is coupled with the individual NMT state machine states. This procedure is advantageous in that the state can be described in great detail. The NMT state machine is defined by the standard CiA 301. There is a good deal of secondary literature available with hints and examples to help deepen your understanding.

This section provides a general description of the structure of an application. The application is divided into numbered areas. The following sections containing descriptions of individual modules make references to these areas in order to specify the positions that must be adapted for integration of the desired module or CANopen services.

2.6.1 Selecting the required modules and configuration

When creating a CANopen device, various CANopen functions and properties are required for object entries. When you acquire a CANopen Library, the parameter of supported services is defined and cannot be modified. However when integrating the CANopen Code, the selection of services is configurable and can be adapted to application requirements.

Services are encapsulated in the modules within the CANopen stack. The following overview shows which module is required by the respective CANopen service. When using the source code, the required modules must be referenced during code generation and the appropriate settings made in file **CopCfg.h** (*refer to section 2.11*). Modules that are listed as base modules always have to be referenced during code generation. Optional modules can be left out if the service they support is not required.

²³: When using the 'debug' version various verification tests are performed and in case of an error the corresponding *PRINTF()* output will be generated.

Service/Function	Module	Category
Initializing CANopen	cmmmain.c	Mandatory module
Managing of PDOs	pdo.c or pdostc.c	optional
SYNC producer	pdosync.c ¹⁾	optional
SYNC consumer	pdosync.c ¹⁾	optional
SDO server	sdocomm.c	Mandatory module
SDO client	sdoc.c	optional
CRC calculation for SDO block transfer	sdocrc.c	optional
Heartbeat producer	hbp.c	optional
Heartbeat consumer	hbc.c	optional
Emergency producer	emcp.c	Mandatory module
Emergency consumer	emcc.c	optional
Life guarding master	nmtm.c	The module Nmtm.c and Nmts.c should always be used in an either-or fashion.
Life guarding slave	nmts.c	
Node guarding master	nmtm.c	
Node guarding slave	nmts.c	
LSS slave	lsslv.c	optional
LSS master	lssmst.c	optional
Creating communication objects for message transmission	cob.c	Mandatory module
Functions for access to the object entries	obd.c	Mandatory module
NMT state machine	nmt.c	Mandatory module
High Precision Time Stamp producer	hpt.c	optional
High Precision Time Stamp consumer	hpt.c	optional
Time Stamp producer	tso.c	optional
Time Stamp consumer	tso.c	optional
Functions for accessing machine specific data formats for the given microcontroller 'Xxx'	amiXXX.c	Mandatory module
Driver for the applicable CAN controller (Xxx) or operating system	cdrvXXX.c	Mandatory module
Interface functions for adapting the hardware-specific CAN controller connections	cciXXX.c	Mandatory module for any CDRV modules
Baud rate table containing the supported baud rates	bditabXXX.c	Mandatory module

Table 17: Guide for selecting the required software modules

- 1) Do not add these files directly to your project.

Modules in the CCM layer are optional except for module **CcmMain.c**. The modules support the user during configuration of the application. The user has to decide which modules to include. An emergency producer is always supported, even if this service is optional according to the standard CiA 301. However, practical application has shown that for diagnosis of an error in an application, this service must be used.

The amount of supported services and protocols within a module can be further reduced (*refer to section 2.11*). This is particularly interesting if very little code and data memory is available on the target. Additional settings must be made in the file **CopCfg.h**. The CANopen stack is implemented independently of any specific CAN controller. For connection of a CAN controller, the specific driver module **CdrvXxx.c** and possibly another module **CciXxx.c** must be included. The module **CciXxx.c** is required if a standalone controller can be connected to a microcontroller in different ways.²⁴ Additional information is available in the manual "CAN Drivers" (*L-1023*).

The baud rate table contains values for various baud rates for the baud rate registers of the CAN controller (e.g. BTR0 and BTR1 in NXP SJA1000). These values are calculated based on the clock frequency of the CAN controller and not the crystal or oscillator frequency. The clock frequency of the CAN controller is usually determined by dividing or multiplying the oscillator frequency of the CAN controller or microcontroller.

2.6.2 Sequence of a CANopen application

A CANopen application has the following cycle in principle:

- Initializing the hardware
- Creating the data structure (object dictionary, Tables, Structures, Variables, Instances) and linking the configured modules configuration of node numbers
- Initialization of services (communication parameters, creating communication objects)
- Processing CANopen and execution of service demands from the application.
- Closing the CANopen layer, if necessary

²⁴: Connection to an INTEL 82C527 CAN controller can be achieved via both serial or parallel interface.

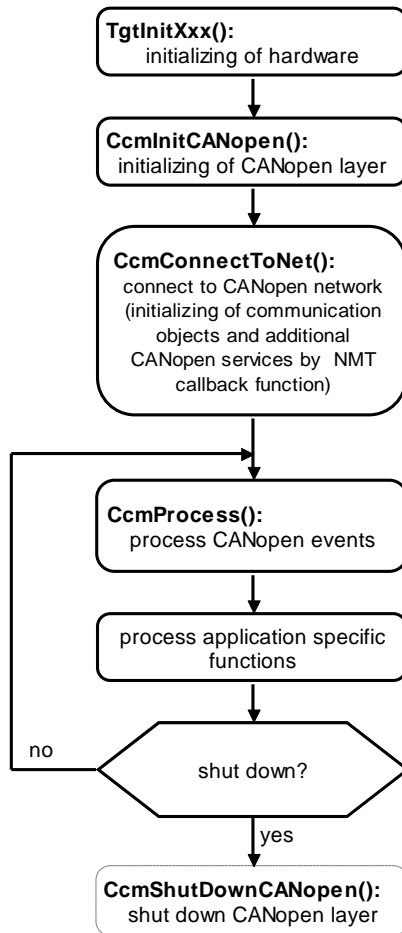


Figure 18: Sequence of a typical CANopen application

2.6.2.1 Initializing the hardware

Before the CANopen layer is initialized, the hardware must be initialized by the application. To function correctly the CANopen requires a time basis, generated in $100\mu\text{s}$, as well as an interface in the debug version for the output of error messages. If an error is discovered based on a faulty configuration or parameterization, then the CANopen layer will call standard C-function `printf` in some cases. The output of the serial data stream to a terminal may need to be adapted for the target.

The global interrupt of the microcontroller is to be released, and the CAN controller's service routine included (which involves setting the interrupt vector and the interrupt priority). Upon delivery, **target.c** files for various target platforms are included with the CANopen Source Code. There are functions in these files for initialization of a timer, the serial interface as well as for release of the global and CAN controller specific interrupt.

Examples for the hardware initialization:

```

void main (void)
{
...
    // disable global interrupt
    TgtEnableGlobalInterrupt (FALSE);

    // init target (timer, interrupts, ...)
    TgtInit ();           // init general
    TgtInitSerial ();    // init serial interface
    TgtInitTimer ();    // init system time
    TgtInitCanIsr ();   // init CAN controller interrupt

    // enable global interrupt
    TgtEnableGlobalInterrupt (TRUE);

...
}

```

When using an operating system, the hardware is usually initialized by the operating system. Functions may be necessary for the initialization of the operating system.

2.6.2.2 Initializing the CANopen layer and creating the data structures

Each module in the CANopen stack or CDRV layer (CAN driver `cdrvXXX.c`) contains a function for the initialization and parameterization of the module. The Init function must be executed for each instance. This step is required in order to correctly process additional functions within the module.

The function `CcmInitCANopen` executes the basic initialization of the CANopen layer. The Init functions of the individual modules are called within this function. This provides the conditions necessary to link application variables (i.e. for storing process data) with the CANopen layer.

Example for initialization of the CANopen layer:

In the following example, initialization of the CANopen layer of a CANopen device is prepared and executed with an instance. The node contains the node number 1, a baud rate of 1 Mbit/s is selected. The clock speed for the controller is 10 MHz for a CPU frequency of 20 MHz. When selecting the baud rate table, it is important to be sure that the listed clock frequency refers to the clock frequency of the CAN controller and not the oscillator frequency of the CAN controller or the CPU. For microcontrollers with an integrated CAN controller or for stand alone CAN controllers, the clock speed can usually be determined by dividing or multiplying the oscillator frequency.

```

#define NODE_ID      0x41           // Node ID is 0x41

// define index to baud rate table for 1 Mbit/sec
#define BAUDRATE     kBd1Mbaud

// define the baud rate table for 10MHz CAN controller clock
#define CDRV_BDI_TABLE_PTR    awCdrvBdiTable10[9]
#define CDRV_BDI_TABLE_SIZE  sizeof(awCdrvBdiTable10)

```

Each CAN identifier can receive. The parameters are stored in a `tCcmInitParam` structure declared as "const". The base address of the CAN controller is entered in the structure by calling `TgtGetCanBase`. A function is defined (`TgtEnableCanInterrupt1`) through the application, which inhibits or releases the CAN controller interrupt. A callback function (`AppCbNmtEvent`) is defined for processing the state changes of the NMT state machine. The function `ObdInitRam`, for initialization of the internal data structures in OD, always has to be entered.

```
CONST tCcmInitParam ROM CcmInitDefaultParam_g =
{
    NODE_ID,                // node id
    BAUDRATE,               // index to baud rate
    CDRV_BDI_TABLE_PTR,    // baud rate table
    CDRV_BDI_TABLE_SIZE,   // size of baud rate table in bytes
    0xFFFFFFFFL,           // Acceptance Mask Register
    0x00000000L,           // Acceptance Code Register
    {{0}},                 // CAN controller address
    TgtEnableCanInterrupt1, // function pointer to
                          // enable CAN interrupt
    AppCbNmtEvent,         // pointer to NMT-Callback
                          // function
    ObdInitRam              // init function for OD
};
```

In this example all entries for the structure are fixed and cannot be changed during runtime. Therefore the structure is stored in the ROM. If the node address or baud rate has to be changed or configured with a DIP switch during runtime, then the structure must be stored in RAM, so that the entries (`m_bInitNodeId`, `m_BaudIndex` etc.) can be modified by the application.

By calling the function `CcmInitCANopen`, the CANopen layer is initialized. The first call of `CcmInitCANopen` is always performed with the parameter `kCcmFirstInstance`. This causes the function to delete the internal instance table.

```

tCcmInitParam MEM CcmInitParam_g;

void main (void)
{
...
    // enable global interrupt
    TgtEnableGlobalInterrupt (TRUE);

    // copy default values to RAM
    CcmInitParam_g = CcmInitDefaultParam_g;

    // set address auf CAN-Controller 1 to tCdrvHwParam
    // (tCdrvHwParam is a UNION, therefore the address cannot be
    // set as const by compiler it must set by user)
    CcmInitParam_g.m_HwParam.m_McIoParam.m_pbBaseAddr =
        TgtGetCanBase (1);

    // initialize first instance of CANopen
    Ret = CcmInitCANopen (&CcmInitParam_g, kCcmFirstInstance);
    if (Ret != kCopSuccessful)
    {
        goto Exit;
    }

    Ret = CcmConnectToNet ();
    if (Ret != kCopSuccessful)
    {
        goto Exit;
    }

    ...

Exit:
    ...
}

```

Now the object dictionary is created, the entries initialized with default values (default values can be provided when the object dictionary is defined). However, object dictionary entries are not linked to the application. NMT state remains in Initialisation.

By calling **CcmConnectToNet** the CANopen Stack follows additional initialisations whereby NMT callback function **AppCbNmtEvent** is called with events. Here application may implement further initialisations and configurations of the CANopen stack. After calling **CcmConnectToNet** the CANopen stack is in NMT state Pre-operational and Bootup message has been sent (if NMT slave is implemented).

2.6.2.3 Node number configuration with LSS

When using the LSS service for configuring a node number, it is important to be sure to execute the LSS state machine before switching from NMT state Initialisation to Pre-operational, if the node number is invalid.

If the application still has no valid node numbers following execution of **CcmInitCANopen** (according to LSS specification CiA DS-305 V1.01, 0xFF is defined as an invalid node number), then the function **CcmProcessLssInitState** must be called cyclically in a loop. CANopen will wait until a valid node number has been initialized via the LSS service before doing this. Once this has occurred, then the function will return a value not equal to *kCopLssInvalidNodeId*. Now the cyclical loop can be ended and **CcmConnectToNet** can be called. The NMT state machine is then started with **CcmConnectToNet**. While this called is performed the NMT callback function within the application is called with various events. Information on what needs to be done within these events is provided in *section 2.6.2.4*.

Example:

```
...
Ret = CcmInitCANopen (&CcmInitParam_g, CcmFirstInstance);
if (Ret != kCopSuccessful)
{
    goto Exit;
}

...
// run LSS init state process until NodeId is valid
do
{
    Ret = CcmProcessLssInitState ();

} while (Ret == kCopLssInvalidNodeId);
...

Ret = CcmConnectToNet ();
if (Ret != kCopSuccessful)
{
    goto Exit;
}
```

If the node number is modified again during the cyclical execution of **CcmProcess**, then a re-initialization of the CANopen layer will be performed automatically (in the CCM module). When this occurs, the events *kNmtEvResetNode*, *kNmtEvRestCommunication* and *kNmtEvEnterPreOperational* will be registered again in the NMT callback function of the application.

2.6.2.4 Initializing services and communication objects, service execution

In the previous step, the basic data structures were created and initialized. The CANopen device contains a valid node number. The step that follows now links the application variables to the entries in the object dictionary and initializes the services and communication objects for the data transfer. Thus the functions to be executed are assigned the states within the NMT state machine.

After the function **CcmlnitCANopen** has been executed, the CANopen device will be in the NMT state machine's *initialisation* state.

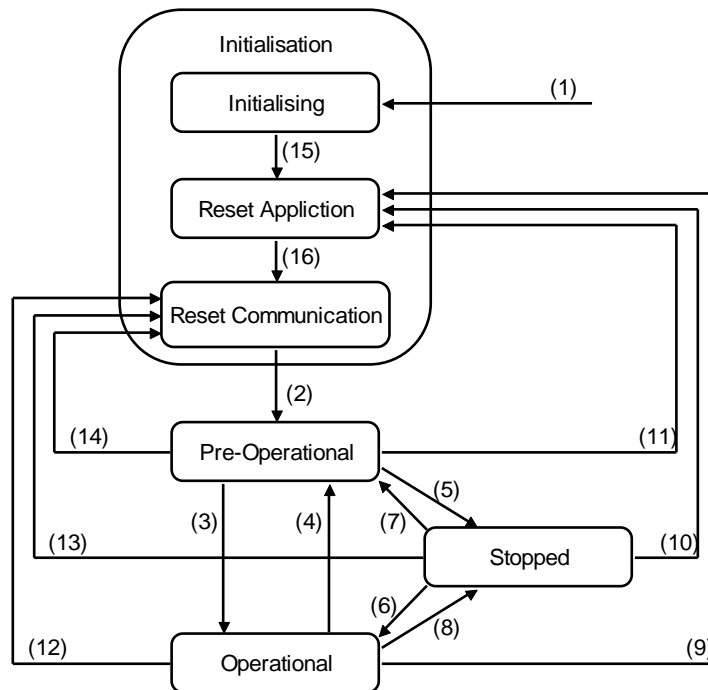


Figure 19: NMT state machine according to CiA 301 V4.02

	CANopen events	Callback events
(1)	Power-on or hardware reset	<i>kNmtEvEnterInitialising</i>
(2)	automatic change into Pre-operational state after completion of Initialisation	<i>kNmtEvEnterPreOperational</i>
(3), (6)	NMT command: start remote node	<i>kNmtEvEnterOperational</i>
(4), (7)	NMT command: enter pre-operational	
(5), (8)	NMT command: stop remote node	<i>kNmtEvEnterStopped</i>
(9), (10), (11)	NMT command: reset node	<i>kNmtEvResetNode</i>
(12), (13), (14)	NMT command: reset communication	<i>kNmtEvPreResetCommunication</i> <i>kNmtEvResetCommunication</i> <i>kNmtEvPostResetCommunication</i>
(15)	automatic change into RESET-APPLICATION state after completion of INITIALIZING	<i>kNmtEvResetNode</i>
(16)	automatic change into RESET-COMMUNICATION state after RESET-APPLICATION finished	<i>kNmtEvPreResetCommunication</i> <i>kNmtEvResetCommunication</i> <i>kNmtEvPostResetCommunication</i>

Table 18: NMT state machine explanation (list of events and commands)

According to the standard CiA 301 the following services are to be supported in the various NMT states:

Communication object	Initialisation	Pre-operational	Operational	Stopped
PDO			X	
SDO		X	X	
SYNC		X	X	
Time stamp		X	X	
Emergency		X	X	
Boot-Up	X			
NMT		X	X	X

Table 19: Supported communication objects in various NMT states [4]

The function **CcmConnectToNet** starts the execution of the NMT state machine with the state *INITIALIZING*. After a state has been closed, the state machine will shift to the next state on its own until reaching the state *Pre-operational*. The function **CcmConnectToNet** will then return. During execution of the individual states respective events, the modules of the CANopen stack will be called repeatedly over the **XxxNmtEvent** function. Likewise a call will be performed for the application's NMT callback function **AppCbNmtEvent**, if a function has been parameterized (entry *m_fpNmtEventCallback* of the structure *tCcmInitParam*). When an NMT callback function is called, the NMT event is given as a parameter (event will be handed over as parameter, refer to Table 18).

The function **AppCbNmtEvent** is called as the last function within the execution sequence of an NMT state's **XxxNmtEvent** functions, allowing previously set standard values to be modified as needed for an application.

In the following examples the function **AppCbNmtEvent** is called as the application's NMT callback function. The examples are based on the condition that only one instance was configured. When multiple instances are used then the instance parameter must be completed.

State Initialisation:

This state is only executed one time following a power-on or reset. In this state the optional modules' Init functions of CCM layer (such as **CcmInitLgs**) must be executed. In this state all application variables have to be linked to the variable entries of the object dictionary. After this is finished, the state machine automatically goes into the *reset application* event.

Example:

```
tCopKernel PUBLIC AppCbNmtEvent (tNmtEvent NmtEvent_p)
{
tCopKernel Ret = kCopSuccessful;

    // which event was called?
    switch (NmtEvent_p)
    {
        // after power-on link all variables with OD
        case kNmtEvEnterInitialising:

            // linking of variables for CANopen with OD
            Ret = CcmDefineVarTab (aVarTab_g,
                sizeof (aVarTab_g) / sizeof (tVarParam));

            break;

        ...
    }
}
```

Sub-state reset application:

In this event all manufacturer specific objects (from 0x2000 to 0x5FFF) and all device specific objects (starting at 0x6000 up to 0x9FFF) have been reset to their power-on values by the CANopen stack. Power-on value refers to the default value from the object dictionary or the last value saved in the non-volatile memory. The application can overwrite these values within this state if necessary.

Example:

```
tCopKernel PUBLIC AppCbNmtEvent (tNmtEvent NmtEvent_p)
{
tCopKernel Ret = kCopSuccessful;

    // which event is called?
    switch (NmtEvent_p)
    {
        case kNmtEvResetNode:

            // reset process vars
            wDigiOut = 0;
            ...
            break;
```

Sub-state reset communication:

Here all communication parameters (starting at 0x1000 to 0x1FFF) are reset to their power-on²⁵ values by the CANopen stack. Power-on value refers to the default value from the object dictionary or the last value saved in the non-volatile memory.

The communication objects for all modules in the CANopen stack are created. The application can now redefine all PDOs. With this, all default settings are overwritten. The state machine changes automatically to *Pre-operational* state after completion. A CANopen slave signals this state transition by sending a BOOTUP message.

Example:

```
tCopKernel PUBLIC AppCbNmtEvent (tNmtEvent NmtEvent_p)
{
tCopKernel Ret = kCopSuccessful;

    // which event is called?
    switch (NmtEvent_p)
    {
        // reset all communication objects (0x1000-0x1FFF)
        case kNmtEvResetCommunication:
            Ret = CcmDefinePdoTab ( (tPdoParam GENERIC*) &aPdoTab_g[0],
                sizeof (aPdoTab_g) / sizeof (tPdoParam));
            break;
```

²⁵: The power-on values are the last values stored in the object 0x1010 (Save Parameters), in as far as they are not reset to their default values with the object 0x1011 (Restore Parameters). It is up to the user to arrange the upload of object dictionary entries into a non-volatile memory. The user is supported thereby by module CcmStore.c.

Dissenting from the NMT state machine in two additional states were implemented within this state:

Callback event kNmtEvPreResetCommunication:

- CANopen stops active services and deletes its communication objects (COBs)
- Callback event kNmtEvResetCommunication is called next

Callback event kNmtEvResetCommunication:

- CANopen resets communication parameters
- Callback event kNmtEvPostResetCommunication is called next

Callback event kNmtEvPostResetCommunication:

- CANopen transmits bootup message to CAN bus and changes NMT state to pre-operational
- Callback event kNmtEvEnterPreOperational is called next

In the state *PRE-RESET-COMMUNICATION* all active services are ended and its communication objects are deleted. In the state *POST-RESET-COMMUNICATION* the transfer of the BOOTUP message is initiated, whereby a CANopen slave signals that the initialization is complete. The state machine changes to *Pre-operational* state.

State Pre-operational:

In this state communication per SDO is possible. life guarding, node guarding or heartbeat is executed if these services were configured by the application. With the help of SDOs, communication parameters and mapping parameters can be modified for PDOs over the CAN bus. The CANopen device switches to state *Operational* after receipt of the NMT *start remote node* from a NMT master²⁶ or after calling the function **CcmBootNetwork** respectively **CcmSendNmtCommand**(0x00, kNmtCommStartRemoteNode).

After the execution of this event the function **CcmConnectToNet** is ended. State changes are now initiated upon receipt of NMT commands. The processing occurs within the function **CcmProcess**.

The function **CcmProcess** must be called in a cyclical loop. The more often it is called, the more stable the CANopen layer's reactions will be to time events.

Within the function **CcmProcess**, CAN messages are evaluated first and assigned to the corresponding internal CANopen modules. If an event occurs that is important for the application, then a callback function will be called. Most of these callback functions are located in the CCM module or are components of the application and can therefore be adapted by the user. Furthermore, the function **CcmProcess** tests a few time cycles, for which a CAN message may have to be sent under certain circumstances. For example, PDOs may be sent following completion of the event timer. Likewise an SDO abort is sent if the SDO server expects a message from the SDO client during a segmented transfer but does not receive one.

²⁶ : For network applications where no NMT master is present changing to *Operational* state can be forced by calling the function **NmtExecCommand** (kNmtCommEnterOperational).

State Operational:

The transmission from *Pre-operational* to *Operational* state generates a transfer of all asynchronous TPDOs. In this state PDOs are transferred if an event occurs (such as event timer expired, SYNC message received, modification of process variables). If PDOs are received, then their data is put into the OD and the application will be notified by calling the corresponding callback function containing applicable parameters.

State Stopped:

In this state the execution of all services are stopped with the exception of NMT services (this also includes node guarding and heartbeat).

2.6.2.5 Shutting down a CANopen application

The CANopen application is closed by executing the function **CcmShutDownCANopen**. This function calls the function **XxxDeleteInstance** for each module that is configured in the CANopen stack. The modules finish their services and delete the communication objects. The data structures of the CANopen layer are invalid after the function **CcmShutDownCANopen** has been executed.

This document has been truncated!

If you wish to receive a complete copy of this document
please contact us via e-mail:
support@systemec-electronic.com

4 Notes on CANopen certification

For CANopen certification with CiA, the following should be noted:

- Only a device can be certified and not software

The CANopen stack was certified with the CANopen-Chip from SYS TEC electronic GmbH.

Certificate No.: CiA200002-301V30/11-013

- Thus we can demonstrate that certification with our CANopen stack is possible.

However, certification also depends on a number of factors, that we cannot influence directly.

Therefore please note the following:

- The entries in the OD must match those in the *.EDS file. This effects above all the device name (Index 0x1008, the hardware and software version (Index 0x1009 or 0x100A) etc.
- The number of PDOs must match the PDOs actually present in the OD
- All indices that are present in the software must also be entered in the EDS file. There can be no hidden entries.
- The entries in Index 0x100C and 0x100D (life guarding) must have a default setting of zero.
- The Index 0x1003, Sub-index 0 can only be written to with a 0 and then the error field has to be erased. Writing a number that is greater than 0 will result in an error.
- The mapping parameter sub-indexes 0 (e.g. 0x1600,0, 0x1601,0, 0x1A00, 0 etc.) can be written with values up to 64 max. If the maximum value is exceed an error message will result. Since our CANopen software supports byte-mapping in its default setting, all values >8 are rejected.
- It must always be possible to answer RTR-queries sent to the node (regardless of TxType).

If the criteria in aforementioned points are met, then certification should be easy.

Note:

We verify our software ourselves with the current version of the CiA Conformance Test Tool. We can also perform pretests of customer devices in house.

5 Index

AMI.....	321	CiA-302-7	87
AMI Interface	321	CiA-303-3	257
Application-specific layer	36	COB callback function.....	212, 216
Big Endian	321	communication object	212
bit rate table.....	317	Communication parameters	153, 157
BOOTUP	25	Communication profile	29
callback function		Configuration	
AppCbNmtEventUnfiltered.....	77	CCM_DR303_USE_BICOLOR_LED	143
AppCbUnknownCobId	76	CCM_MODULE_INTEGRATION78, 89, 93, 103, 139, 257	
CcmCbEmccEvent	116	CCM_PROCESS_RECV_COUNT	258
CcmCbEmcpEvent	119	CCM_STORE_FILE_SYSTEM	62, 258
CcmCbError	71, 74	CCM_USE_PDO_CHECK_VAR.....	258
CcmCbHbcEvent	124	CCM_USE_STORE_RESTORE.....	95, 257
CcmCbLgsEvent.....	94	CDRV_CAN_SPEC.....	261
CcmCbLssmEvent.....	150	CDRV_IDINFO_ALGO	262
CcmCbLsssEvent	74	CDRV_IDINFO_ENTRIES	263
CcmCbNmtEvent	70	CDRV_IMPLEMENT_RTR	266
CcmCbNmtmEvent.....	107	CDRV_MAX_INSTANCES	259
CcmCbRestore	98	CDRV_MAX_RX_BUFF_ENTRIES_HIGH	292
CcmCbStore.....	97	CDRV_MAX_RX_BUFF_ENTRIES_LOW	292
CcmCbStoreLoadObject.....	100	CDRV_MAX_RXPOLLING	265
CcmCbSyncReceived	111, 113	CDRV_MAX_TX_BUFF_ENTRIES_HIGH.....	292
tCcmCbUnknownCobId	76	CDRV_MAX_TX_BUFF_ENTRIES_LOW.....	292
tCcmNmtEventCbUnfiltered	77	CDRV_TIMESTAMP	263
CDRV_USE_BASIC_CAN.....	260	CDRV_USE_DELETEINST_FUNC	265
CDRV_USE_DELETEINST_FUNC	265	CDRV_USE_ERROR_ISR	264
CDRV_USE_ERROR_ISR	264	CDRV_USE_HIGHTHRESHOLD	261
CDRV_USE_HIGHTHRESHOLD	261	CDRV_USE_HPT.....	265
CDRV_USE_IDVALID	261	CDRV_USE_IDVALID	261
CDRV_USE_NO_ISR.....	264	CDRV_USE_NO_ISR.....	264
CDRV_USE_NO_RXBUFF	264	CDRV_USE_NO_RXBUFF	264
CDRV_USE_NO_TXBUFF	263	CDRV_USE_NO_TXBUFF	263
CDRV_USE_SETBAUDRATE_FUNC	267	CDRV_USE_SETBAUDRATE_FUNC	267
CDRV_USED_CAN_CONTROLLER.....	260	CDRV_USED_CAN_CONTROLLER.....	260
CDRVLIN_USE_NEW_HWPARAM_API	267	CDRVLIN_USE_NEW_HWPARAM_API	267
CDRVWIN_USE_CYCLIC_TX_INTERFACE	266, 269	CDRVWIN_USE_CYCLIC_TX_INTERFACE	266, 269
COB_IMPLEMENT_SET_PARAM	269	COB_IMPLEMENT_SET_PARAM	269
COB_MAX_RX_COB_ENTRIES	212, 292	COB_MAX_RX_COB_ENTRIES	212, 292
COB_MAX_TX_COB_ENTRIES	212, 292	COB_MAX_TX_COB_ENTRIES	212, 292
COB_MORE_THAN_128_ENTRIES	267	COB_MORE_THAN_128_ENTRIES	267
COB_MORE_THAN_256_ENTRIES	268	COB_MORE_THAN_256_ENTRIES	268
COB_SEARCHALGO.....	268	COB_SEARCHALGO.....	268
COB_USE_ADDITIONAL_API	269	COB_USE_ADDITIONAL_API	269
COB_USE_CB_UNKNOWN_COBID.....	270	COB_USE_CB_UNKNOWN_COBID.....	270
COB_USE_CYCLIC_TX_INTERFACE	269	COB_USE_CYCLIC_TX_INTERFACE	269
COB_USE_RTR_CONSUMER	268	COB_USE_RTR_CONSUMER	268
COP_CB_DIRECT_CALL	256	COP_CB_DIRECT_CALL	256
COP_MAX_INSTANCES	252	COP_MAX_INSTANCES	252
COP_USE_CDRV_FUNCTION_POINTER	62, 252	COP_USE_CDRV_FUNCTION_POINTER	62, 252
COP_USE_DELETEINST_FUNC.....	66, 255	COP_USE_DELETEINST_FUNC.....	66, 255
COP_USE_OPERATION_SYSTEM	255	COP_USE_OPERATION_SYSTEM	255
COP_USE_SMALL_TIME	255	COP_USE_SMALL_TIME	255
COP_USE_TGTOS_API	255	COP_USE_TGTOS_API	255
DEF_DEBUG_LVL	250	DEF_DEBUG_LVL	250
EMCC_MAX_CONSUMER	292	EMCC_MAX_CONSUMER	292
EMCP_CHECK_COBID_ORDER.....	278	EMCP_CHECK_COBID_ORDER.....	278
EMCP_ENABLE_ERROR_WRITE	278	EMCP_ENABLE_ERROR_WRITE	278
EMCP_USE_EVENT_CALLBACK	277	EMCP_USE_EVENT_CALLBACK	277
LSS-Callback-Function	151		
Master-Callback-Function	108		
NMT-Commands	77		
Unknown CAN-messages	76		
CAN bit rate	317		
CAN driver	317		
selection	315		
CAN ERROR LED	139, 142		
CAN RUN LED	138, 141		
CANopen stack.....	34		
CANopen stack configuration	250		
CANopen stack functions	155		
Ccixxx.c	315		
CCM	36		
cdrvtgt.h.....	317		
Cdrvxxx.h.....	315		
Certification.....	325		
CiA 303-3.....	138		
CiA-302	257		

EMCP_USE_INHIBIT_TIME	278	FUNCTION_CALL_TIME	275
EMCP_USE_PREDEF_ERROR_FIELD	277	FUNCTION_NO_CONTROL	275
HBC_IGNORE_BOOTUP	291	kCobTypForceRmtRecv	214
HBC_MAX_EMCY_VALUES	259	kCobTypForceSend	214
HBC_USE_ADDITIONAL_API	291	kCobTypRecv	214
LSSM_CONFIRM_TIMEOUT	291	kCobTypRmtRecv	214
LSSM_PROCESS_DELAY_TIME	290	kCobTypRmtSend	214
NMTM_MAX_SLAVE_ENTRIES	292	kCobTypSend	214
NMTS_USE_CB_MONITOR_ALL_COMMANDS	277	kDr303ErrorBusoff	142
NMTS_USE_LIFEGUARDING	93, 276	kDr303ErrorControlEvent	142
NMTS_USE_NODEGUARDING	276	kDr303ErrorLssProcess	142
OBD_CALC_OD_SIGNATURE	271	kDr303ErrorSyncTimeout	142
OBD_CHECK_FLOAT_VALID	271	kDr303ErrorWarningLimit	142
OBD_CHECK_OBJECT_RANGE	270	kDr303NoError	142
OBD_IMPLEMENT_ARRAY_FCT	273	kDr303RunDeviceOperational	141
OBD_IMPLEMENT_DEFINE_VAR	274	kDr303RunDevicePreOperational	141
OBD_IMPLEMENT_INIT_MOD_TAB	274	kDr303RunDeviceStopped	141
OBD_IMPLEMENT_PDO_FCT	273	kDr303RunLssProcess	141
OBD_IMPLEMENT_READ_WRITE	273	kDr303RunRese	141
OBD_INCLUDE_A000_TO_DEVICE_PART	272	kLssmCmdInquireNodeld	148
OBD_SUPPORTED_OBJ_SIZE	270	kLssmCmdInquireProductCode	148
OBD_USE_DYNAMIC_OD	271	kLssmCmdInquireRevisionNr	148
OBD_USE_STRING_DOMAIN_IN_RAM	271	kLssmCmdInquireSerialNr	148
OBD_USE_USTRING	274	kLssmCmdInquireVendorId	148
OBD_USE_VARIABLE_SUBINDEX_TAB	272	kLssmEvActivateBitTiming	150
OBD_USER_OD	207	kLssmEvActivateBusContact	150
PDO_CHECK_COBID_ORDER	283	kLssmEvDeactivateBusContact	150
PDO_DISABLE_FORCE_PDO	281	kLssmEvIdentifyAnySlave	150
PDO_GRANULARITY	279	kLssmEvInquireData	150
PDO_IMPLEMENT_CHECK_VAR	283	kLssmEvModeSelective	150
PDO_MAX_EMCY_VALUES	259	kLssmEvResult	150
PDO_MORE_THAN_255_ENTRIES	284	kLssmEvTimeout	150
PDO_PROCESS_TIME_CONTROL	278	kLssModeConfiguration	145
PDO_USE_ADDITIONAL_API	283	kLssModeOperation	145
PDO_USE_BIT_MAPPING	282	kLssModeSelective	145
PDO_USE_DEF_LINKING_IN_OD	284	kLsssEvActivateBitTiming	152
PDO_USE_DEF_MAPPING_IN_OD	284	kLsssEvActivateBusContact	152
PDO_USE_DUMMY_MAPPING	281	kLsssEvConfigureBitTiming	152
PDO_USE_ERROR_CALLBACK	281	kLsssEvConfigureNodeld	152
PDO_USE_EVENT_TIMER	279	kLsssEvDeactivateBusContact	152
PDO_USE_MPDO_DAM_CONSUMER	282	kLsssEvEnterConfiguration	152
PDO_USE_MPDO_DAM_PRODUCER	282	kLsssEvEnterOperation	152
PDO_USE_MPDO_SAM_CONSUMER	282	kLsssEvPreResetNode	152
PDO_USE_MPDO_SAM_PRODUCER	282	kLsssEvSaveConfiguration	152
PDO_USE_REMOTE_PDOS	280	kNmtCommEnterOperational	218
PDO_USE_STATIC_MAPPING	89, 281	kNmtCommEnterPreOperational	218
PDO_USE_SYNC_CONS_COUNTER	285	kNmtCommEnterStopped	218
PDO_USE_SYNC_PDOS	280	kNmtCommInitialize	218
PDO_USE_SYNC_PROD_COUNTER	284	kNmtCommResetCommunication	218
PDO_USE_SYNC_PRODUCER	280	kNmtCommResetNode	218
PDO_VARCB_BEFOR_ENCODE	282	kNmtCommStartRemoteNode	218
SDO_BLOCKSIZE_DOWNLOAD	286	kNmtCommStopRemoteNode	218
SDO_BLOCKSIZE_UPLOAD	286	kNmtErrCtrlEvBootupReceived	107
SDO_BLOCKTRANSFER	285	kNmtErrCtrlEvHbcConnected	124
SDO_CALCULATE_CRC	163, 176, 287	kNmtErrCtrlEvHbcConnectionLost	124
SDO_MAX_CLIENT_IN_OBD	178	kNmtErrCtrlEvHbcNodeStateChanged	124
SDO_SEGMENTTRANSFER	174, 175, 286	kNmtErrCtrlEvLgConnected	95
SDO_USE_SDO_ROUTER	289	kNmtErrCtrlEvLgLostConnection	95
SDOC_DEFAULT_TIMEOUT	288	kNmtErrCtrlEvLgMsgLost	95
SDOC_NETWORK_RESPONSE_TIMEOUT	88, 289	kNmtErrCtrlEvLgNoAnswer	107
SDOC_USE_NETWORK_INDICATION	87, 186, 289	kNmtErrCtrlEvLgNodeStateChanged	107
SDOR_MAX_ROUTING_ENTRIES	290	kNmtErrCtrlEvLgSuspended	107
SDOR_ROUTER_FORWARDING	290	kNmtErrCtrlEvLgToggleError	107
SDOR_SDO_CLIENT_INDEX	290	kNodeStateInitialisation	206
SDOS_DEFAULT_TIMEOUT	288	kNodeStateOperational	206
SDOS_MULTI_SERVER_SUPPORT	288	kNodeStatePreOperational	206
SDOS_USE_ADDITIONAL_API	287	kNodeStateStopped	206
TARGET_HARDWARE	252, 315, 319	kObdAccVar	205
Constant			
CCM_LSSFLAGS_ALL	148	kObdCommClear	102
CCM_LSSFLAGS_SLAVE_ADDRESS	148	kObdCommCloseWrite	102
CCM_MODULE_DR303_3	139	kObdCommOpenRead	102
CDRV_IDINFO_ALGO_BITFIELD	262	kObdCommOpenWrite	102
CDRV_IDINFO_ALGO_FULLCAN	262	kObdCommReadObj	102
CDRV_IDINFO_ALGO_IDLISTEXT	262	kObdCommWriteObj	102
EMCP_EVENT_ERROR_DELETEALL	119	kObdDirInit	204
EMCP_EVENT_ERROR_LOG	119	kObdDirLoad	204
FUNCTION_BREAK_TIME	275	kObdDirRestore	204
		kObdDirStore	204
		kObdEvAbortSdo	209

kObdEvCheckExist	208	CcmDefineVarTab	67
kObdEvInitWrite	208	CcmEmccDefineProducerTab	114
kObdEvPostRead	208	CcmEnterCriticalSectionPdoProcess	311
kObdEvPostWrite	209	CcmExecNmtCommand	75
kObdEvPreRead	208	CcmGetNmtState	75
kObdEvPreWrite	208	CcmGetNodeld	75
kObdEvWrStringDomain	209	CcmGetSyncCounter	112
kObdPartAll	203	CcmHbcDefineProducerTab	122
kObdPartDev	203	CcmInitCANopen	61
kObdPartGen	203	CcmInitEmcc	114
kObdPartMan	203	CcmInitHbc	121
kObdPartUsr	203	CcmInitLgs	93
kSdocTransferFinished	85	CcmInitNmtm	104
kSdocTransferNotActive	85	CcmInitNmtmEx	108
kSdocTransferRunning	85	CcmInitStore	96
kSdocTransferRxAborted	85	CcmInitSyncConsumer	110, 113
kSdocTransferTxAborted	85	CcmInitSyncConsumerEx	113
LSS_INVALID_NODEID	75, 146, 206	CcmLeaveCriticalSectionPdoProcess	311
OBD_OBJ_SIZE_BIG	270	CcmLockCanopenThreads	299
OBD_OBJ_SIZE_MIDDLE	270	CcmLockedCopyData	300
OBD_OBJ_SIZE_SMALL	270	CcmLssmConfigureSlave	146
SDO_CRC_POLYNOM	287	CcmLssmIdentifySlave	149
SDO_CRC_TABLE	287	CcmLssmInquireIdentity	147
SDO_NO_CRC	287	CcmLssmSwitchMode	144
copcfg.h	317	CcmPdoSendMPDO	239
CRC calculation	176	CcmProcess	70
DAM	282, 327	CcmProcessLssInitState	69
data array	137, 198	CcmReadObject	92
data structures	40, 155	CcmRegisterErrorCallback	72, 74
Development Environment	293	CcmSdocAbort	86
Directory structure	39	CcmSdocDefineClientTab	79
dynamic memory management	319	CcmSdocGetState	84
EMCC callback function	113, 114, 116, 225, 227	CcmSdocIndicateNetwork	87
Emergency consumer	225	CcmSdocStartTransfer	82
Emergency error codes	249	CcmSendEmergency	118
Emergency object	21	CcmSendNmtCommand	105
Emergency producer	228	CcmSendThreadEvent	300, 310
Error callback function	71	CcmShutDownCANopen	66
error codes	241	CcmSignalCheckVar	109
error handling	31	CcmSignalStaticPdo	138
free	319	CcmStoreCheckArchivState	96
Function		CcmStoreRestoreDefault	103
Ccm303InitIndicators	140	CcmTriggerNodeGuard	106
Ccm303ProcessIndicators	140	CcmUnlockCanopenThreads	299
Ccm303SetErrorState	141	CcmWriteObject	92
Ccm303SetRunState	141	CobCheck	215
CcmBootNetwork	134	CobDefine	213
CcmClearPreDefinedErrorField	118	CobProcessRecvQueue	216
CcmConfigEmcp	117	CobSend	215
CcmConfigHbp	125	CobUndefine	214
CcmConfigLgm	108	EmccAddInstance	226
CcmConfigLgs	94	EmccAddProducerNode	228
CcmConfigSyncConsumer	110	EmccDeleteInstance	226, 230
CcmConigSyncProducer	111	EmccDeleteProducerNode	228
CcmConnectToNet	66	EmccInit	225
CcmConvertFloat	135	EmccNmtEvent	227
CcmDefinePdoTab	89	EmccSetEventCallback	227
CcmDefineSlaveTab	104	EmcpAddInstance	229
CcmDefineStaticPdoTab	136	EmcpInit	229
		EmcpNmtEvent	230
		EmcpSendEmergency	231
		HbcAddInstance	232
		HbcDeleteInstance	232
		HbcInit	231
		HbcNmtEvent	233
		HbcSetEventCallback	233
		HbpAddInstance	234
		HbpDeleteInstance	235
		HbpInit	234
		HbpNmtEvent	235
		HbpProcess	236
		NmtExecCommand	75, 217
		NmtmAddSlaveNode	221
		NmtmConfigLgm	222
		NmtmDeleteSlaveNode	221
		NmtmGetSlaveInfo	223
		NmtmProcess	225
		NmtmSendCommand	224
		NmtmTriggerNodeGuard	223

NmtsProcess	219	heartbeat producer	27
NmtsSendBootup	219	heartbeat producer	234
NmtsSetLgCallback	220	Indicator Specification.....	138, 257
ObdAccessOdPart	203	instance handle	46
ObdDefineVar	205	instance pointers	46
ObdGetEntry	200	Intel Format	321
ObdGetNodeId.....	75, 206	Kernel Mode Driver.....	296, 301
ObdGetNodeState	75, 205	layer setting service	22
ObdReadEntry	202	Lgs callback function	93, 94, 220
ObdRegisterUserOd	207	life guarding.....	26
ObdWriteEntry	201	Linux	296
PdoAddInstance	190	Little Endian	321
PdoDefineCallback	192	LSS.....	22
PdoDeleteInstance	191	LSS address.....	145, 147
PdoForceAsynPdo.....	197	LSS master	22, 144
PdoInit	190	LSS mode	23
PdoNmtEvent	191	LSS slave.....	22, 144
PdoProcessAsyn.....	195	Macro	
PdoProcessCheckVar	194	CCM_CONVERT_LSSCMD_TO_LSSFLAG.....	147
PdoProcessSync	196	COP_FREE	319
PdoSend.....	193	COP_MALLOC.....	319
PdoSendMPDO	238	COP_MEMCPY.....	319
PdoSendSync.....	196	COP_MEMSET	319
PdoSetSyncCallback	197	TGT_CONFIG_CANOPEN_LEDS.....	143
PdoSignalDynPdo.....	193	TGT_SWITCH_ERROR_LED.....	139, 143
PdoSignalStaticPdo.....	199	TGT_SWITCH_RUN_LED.....	143
PdoSignalVar.....	194	malloc.....	319
PdoStaticDefineVarField	199	Master callback function103, 104, 106, 107, 225	
SdocAbort.....	185	Master callback function	221
SdocAddInstance.....	178	memory specifier	
SdocDefineClient	179	CONST	314
SdocDeleteInstance.....	178	FAR.....	314
SdocGetState	183	GENERIC.....	314
SdocIndicateNetwork	185	HWACC.....	314
SdocInit	176	LARGE.....	314
SdocInitTransfer	181	MEM.....	314
SdocNmtEvent.....	179	NEAR	314
SdocProcess	184	ROM.....	314
SdocUndefineClient	180	ROM_INIT	314
SdosAbort.....	171	module	
SdosAddInstance.....	167	CavFile	125
SdosDefineServer.....	168	Ccm303.....	138
SdosDeleteInstance.....	167	CcmBoot	134
SdosInit	166	CcmDfPdo.....	89
SdosNmtEvent.....	168	CcmEmcc.....	113
SdosProcess	170	CcmEmcp.....	117
SdosUndefineServer.....	170	CcmFloat.....	135
TgtCalcCrc16.....	164	CcmHbc	121
TgtCanIsrxxx	320	CcmHbp	125
TgtCavCheckValid	133	CcmLgs	93
TgtCavClose.....	129	CcmLss.....	144
TgtCavCreate	127	CcmMain	60
TgtCavDelete.....	128	CcmMPdo	239
TgtCavGetAttrib	132	CcmNmTm	103
TgtCavInit	126	CcmObj	91
TgtCavOpen.....	128	CcmSdoc.....	78
TgtCavRestore.....	131		
TgtCavShutDown.....	126		
TgtCavStore.....	130		
TgtEnableCanInterrupt.....	320		
TgtEnableGlobalInterrupt	319		
TgtGetCanBase.....	320		
TgtGetTickCount.....	320		
TgtInit	319		
TgtInitCanIsr	320		
TgtInitSerial	319		
TgtInitTimer	320		
TgtMemCpy.....	319, 320		
TgtMemSet.....	319, 320		
TgtTimerIsr	320		
GLOBAL.H.....	313		
Hardware-specific layer.....	36		
HBC callback function	121, 124, 231, 233		
heartbeat	26, 27		
heartbeat consumer	28, 231		

CcmSnPdo	109	PDO linking	13, 58, 89
CcmStore	95	PDO mapping	14, 58, 89
CcmStPdo	136	PDO Receive Notification.....	188
CcmSync	109	PDO Remote Frame	280
CDRV	36	PDO Send Notification	188
COB	212	PDO Synchronization.....	110, 111, 196
EMCC	225	PDO transmission.....	14, 188, 193, 194, 195
EMCP	228	pre-defined connection set	25
HBC	231	Process data object	13
HBP	234	process variables.....	153
MPDO	237	PxROS	293
NMT	217	reset communication.....	26, 90
NMTM	221	reset node	90
NMTS	219	return codes	241
OBD	200	SAM	282, 327
PDO	186	SDO	19
PDOSTC	35, 186, 198	SDO abort codes	248
SDOC	171	SDO block transfer	163, 176
SDOS	155	SDO callback function.....	82, 84, 86, 176, 181, 182, 184, 185
TgtCav.....	125	SDO client	171
Motorola Format.....	321	SDO client creation.....	173
MPDO.....	282, 327	SDO client table.....	172
Network Management	25	SDO download	160
NMT		SDO expedited download	174
Initialisation.....	55	SDO expedited upload.....	175
Operational.....	206	SDO segmented download	174
Pre-operational	206	SDO segmented upload.....	175
Stopped.....	206	SDO transfer	159
NMT callback function.....	54, 56, 67, 70, 81, 153, 217	SDO upload.....	162
NMT command	217, 220	SDO_CRC_POLYNOM	163
NMT Error.....	72	SDO_CRC_TABLE	163
NMT state machine	25, 72, 217	SDO_CRC_TARGET	163
node guarding.....	26	SDO_NO_CRC.....	163
Node Number	206	SDOC data structures.....	172
Node State.....	206	SDO-Gateway	87, 185
NTM		Sending PDOs.....	186
Initialisation.....	25, 54, 66	Service data object	19
Operational.....	25, 58	Software structure.....	33
Pre-operational	25, 54, 56, 57, 65, 66, 72		
Stopped	26		
Object callback function.....	91, 97, 98, 157, 173, 201, 202, 207		
Object callback function	165		
object dictionary	30, 43		
configuration	292		
OD for I/O devices	43		
Operating Systems	293		
PDO	13, 186, 257		
PDO callback function.....	90, 188, 192		
PDO configuration.....	57, 89		
PDO error	73		
PDO event time	58, 188, 279		
PDO inhibit time.....	19, 188		

static PDO mapping ...	35, 136, 186, 198, 281
Store callback function	96, 100
Structure	
tCcmlInitParam	61, 305
tCobCdrvFct	253
tCobParam	213
tEmcParam	116
tHbcProdParam	122
tLinuxParam	297
tLssAddress	145
tLssCbParam	151
tLssmBitTiming	147
tLssmIdentifyParam	149
tLssmResult	151
tMPdoParam	238
tNmtmSlaveInfo	224
tNmtmSlaveParam	222
tObdCbParam	208
tObdCbStoreParam	100
tObdVStringDomain	209
tPdoError	73
tPdoParam	90
tPdoStaticParam	137
tSdocCbFinishParam	182
tSdocInitParam	177
tSdocNetworkParam	87
tSdocParam	79, 180
tSdocTransferParam	82, 181
tSdosInitParam	166
tSdosParam	169
tVarParam	67
tWindowsParam	305
SYNC callback function	109, 110, 111, 187, 197
Synchronization object	21
target.c	319
target.h	319
Telegram Table	32
Time stamp object	21
Transmission Protocols	30
Type	
tSdocState	85
tThrdPriority	306
tVxDType	306
User Layer	33
Variable callback function	40, 68, 188
Windows	301

6 Glossary

User Layer:

CiA 301: Definition of communication profile and application layer

Framework:

CiA 302: Framework for programmable CANopen devices

CiA 304: Framework for safety relevant communication

Communication profile Specification of transmission protocols, communication objects, data objects

Device profile Specification of device-specific services and properties

CiA 401 CiA Draft Standard 401

Device profile for generic I/O modules

Object dictionary (OD): The object dictionary (OD) is the main data structure of a CANopen devices for storage of all device data. It serves as a binding element between the application and the communication layer. Any OD entry is address via an index and a sub-index.

Communication object: Object for transmitting data between CANopen devices.

TPDO Communication object for sending process data (Transmit Process Data Object)

RPDO Communication object for receiving process data (Receive Process Data Object)

Tx-Type PDO transmission type. This always corresponds to sub-index 2 of the PDO communication parameter (object index 0x1400 to 0x15FF and 0x1800 to 0x19FF).

MPDO Multiplexed PDO – Enables the transmission of process data in an SDO-like manner. It is possible to transmit data to one or multiple devices simultaneously without having a PDO for each single object.

DAM Destination Address Mode – MPDO mode where the producer addresses the destination object in the consumer's OD.

SAM Source Address Mode – MPDO mode where the producer gives the address of the source object in the local OD. The producer has a Scanner-list containing all the objects to be sent. The consumers have a corresponding dispatcher list. This list connects each producer's source object to a destination object in the consumer's OD.

7 References

- [1] „CANopen User Manual“, Software Manual, SYS TEC electronic GmbH, L-1020e, this manual
- [2] „CAN driver“, Software Manual, SYS TEC electronic GmbH, L-1023e
- [3] „CANopen Objekt Dictionary“, Software Manual, SYS TEC electronic GmbH, L-1024e
- [4] „CANopen - Application Layer and Communication Profile“, CiA¹ 301 Work Draft, Version V4.02.0.72, Juny 2012
- [5] „CANopen - Framework for CANopen Managers and Programmable CANopen Devices“, CiA¹ Draft Standard Proposal 302, V4.1, February 2009
- [6] „CANopen - Interface and Device Profile for IEC 61131-3 Programmable Devices“, CiA¹ Draft Standard 405, V2.0, 21. 05. 2002
- [7] „CANopen - Device Profile for Generic I/O Modules“, CiA¹ Draft Standard 401, V3.0, June 2008

¹ CiA CAN in Automation e.V.

Document: CANopen Software Manual
Document number: L-1020e_14, September 2015

How would you improve this manual?

Did you find any mistakes in this manual? page

Submitted by:

Customer number: _____

Name: _____

Company: _____

Address: _____

Return to:

SYS TEC electronic GmbH
Am Windrad 2
D-08468 Heinsdorfergrund
GERMANY
Fax : +49 (0) 3765 / 38600-214

