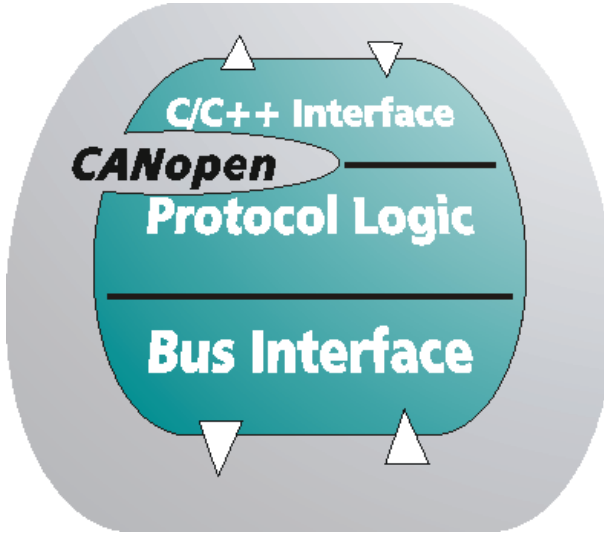
Softing Industrial Automation GmbH
Richard-Reitzner-Allee 6
D-85540 Haar
Tel.: (++49) 89/4 56 56-0
Fax.: (++49) 89/4 56 56-399
http://www.softing.com

Softing CANopen Client API

User Manual



Version 5.17.1
August 2012

Contents

List of figures

List of tables

Preface

About this manual

This user manual is written for users operating the CANopen Client API on Softings CAN interfaces within the operating systems Windows 7, Vista and XP.

It includes the following topics:

- Chapter 1 gives a common introduction about the product and its application.
- In Chapter 2 the installation is described. Helpful notes support the uncomplicated installation. A ‘Quick start’ is included.
- Chapter 3 provides a short introduction into the CANopen network specified in [CiA 301]. Communication objects and protocols are described briefly.
- Chapter 4 describes how to realize the CANopen functionality of Chapter 3 using the CANopen Client API. Application notes, programming sequences and example links are included.
- Chapter 5 provides some helpful programming hints regarding API linking and interrupt.
- The API function reference in alphabetic order can be found in Chapter 6.

In addition to this user manual, always observe the notes contained in file *readme.txt*. This file resides on the disk along with the setup program. The notes contain up-to-date information concerning the present software version.

The hardware description including pin-out configuration and installation is attached to the respective hardware.

1 About the CANopen Client API

1.1 Scope of Application

Nowadays PCs become more and more important as standard components for visualization and controlling of automated machinery. On the other hand CANopen has established a widespread standard in the fieldbus world due to its flexibility and scalability. Fieldbus interfaces are required to connect the PC with the distributed field devices. The rating of the interface regarding data throughput and responsiveness are most important for the reliability and availability of the automated system.

Meeting these requirements the CANopen Client API merges the high-performance of Softings CAN interfaces with the CANopen Master functionality as specified in the CANopen Specification [CiA 301 V4.0.2].

The CANopen Client API is designed to fulfill the customer's requirements regarding, responsiveness, data throughput and easy implementation provided by following main features:

CANopen Stack

- The CANopen Stack runs as a 32bit Windows DLL on Softing's speed-optimized CAN Layer2 API.

Asynchronous User Interface

- Some application commands are executed asynchronously without immediate result information.
- CANopen events to the application are buffered in a FIFO (255 entries) overcoming short breaks without losing data or information.
- Hardware interrupt triggered event evaluation.

Implementation

- 32bit Windows DLL for Windows 7, Vista and XP for 32bit applications. These 32bit applications may as well run on 64bit PCs as the CANopen Client API DLL is able to connect to Softing's 64bit CAN driver.

The CANopen Client API is compatible with the CANopen specification [CiA 301 V 4.0.2]. It comprises the following CANopen specific features:

SDO Manager
- Expedited and segmented down and upload

PDO Manager
- Buffering of up to 512 TPDOs and 512 RPDOs

Node Manager (NMT Master)
- Remote state change of the slaves
- Emergency (EMCY) support
- Error Control (Heartbeat, Node Guarding)
- Boot-up message at start-up

SYNC Manager
- Producing and consuming SYNC messages

1.2 Supported Systems

The CANopen Client API functions are integrated in a 32bit DLL according to Standard C calling convention. Thus, all compilers, measurement tools and visualization systems that are able to provide access to 32bit Windows DLLs may work with this DLL

Some technical hints for implementing the API into the certain systems can be found in section 5.1.

2 Getting Started

2.1 System Requirements

Applying the CANopen Client API your system must meet the following requirements:

- Windows Windows 7, Vista or XP running.
- 150 MByte free space on hard disk
- Softing CAN interface card

2.2 Quick Start

Common installation procedure of the CANopen Client API for PC Windows systems:

1. Run *Setup.exe* in the root directory of the CD and follow the installation dialog.
2. Chose the optional setup component “LeanCANopen” to be installed.
3. Shut down the PC (not required for USB or PCMCIA based interface cards).
4. Insert the CAN interface board into your PC.
5. Restart the PC (not required for USB or PCMCIA based interface cards).
6. If the Hardware Assistant of the windows system comes up follow the windows dialog installing the hardware drivers.

2.3 Installation Test

An installation test program can be found at *[Installation Directory] \LeanCANopen\Win32*.

It is named *InstTest.exe* and involves four sub-tests described in Table 2-1.

While trying to connect to a CAN network operated at 1Mbit/s, the program reports success and/or problems in detail.

Figure 2-1 shows the screen dump of a successful installation test on a hardware interface card which is not connected to a communication partner.

Table 2-1: Installation tests of InstTest.exe

Test	Protocol
1. Hardware access test	Success or Error return code of *CMA_InitializeCard* (see section 6.8)
2. CAN controller access test	Success of CAN channel initialization
3. Interrupt test	Success of interrupt processing including assigning and detecting a WIN32 interrupt event. (If unsuccessful installation test is continued by polling the interface.)
4. CANopen Master firmware test	Success of start-up and shutdown of the CANopen Client including network boot-up. A boot-up message with CAN ID = 1919 is sent. If no communication partner is connected to the CANopen Client or if the network is operated at a speed other than 1Mbit/s, the test states an error accessing the CAN bus.

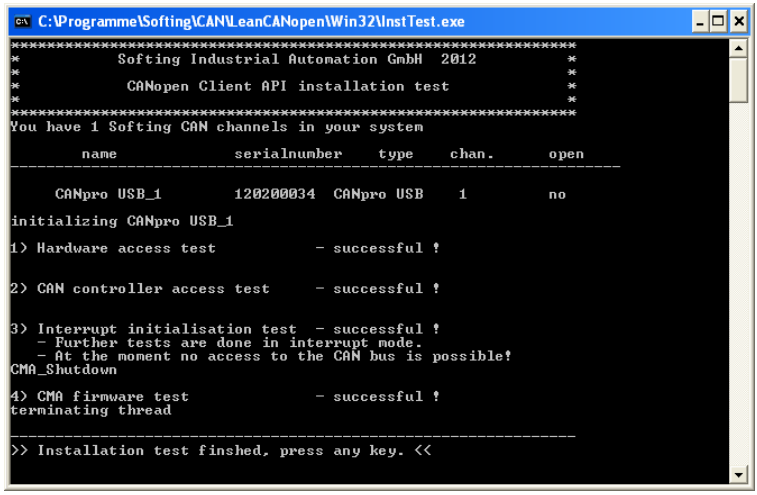


Figure 2-1: Installation test screen

2.3.1 Uninstall Support

After successful installation the *'CANopen Client API'* can be uninstalled by modifying the *'Softing CAN Drivers and Software'* entry in the installed program list of Windows.

2.4 Hardware Driver Notes

2.4.1 Driver Files

The hardware access of the CANopen Client API (*CANopenL2.dll*) requires properly installed hardware drivers and CAN Layer2 API. These components are mandatory items of the software setup.

2.4.2 Compatibility Note

The hardware drivers also provide hardware access for other CAN applications, e.g.:

- CAN L2 API V5.x
- DeviceNet API V2.x

The CANopen Client API runs with the supplied hardware driver version (see *readme.txt*). It will probably run with future versions of the hardware.

If an older hardware driver version is already installed the driver should be updated by the setup program. The new driver version runs also together with the older CAN software listed above.

3 About CANopen [CiA 301]

3.1 Common

3.1.1 CANopen Network

CANopen provides standardized communication mechanisms and device functionality on basis of the **C**ontrol **A**rea **N**etwork (ISO 11898). Since CANopen comprises real-time data exchange as well as peer-to-peer communication it became a common network in automated machinery.

CANopen is defined in the application layer of the ISO-OSI Reference Model as shown in Figure 3-1. The communication profile [CiA 301] specifies communication objects and services for data exchange, device configuration and network management. The CANopen devices are standardized by their related device profiles [CiA 40x]. All standards are maintained by the user and manufacturer group CAN in Automation (CiA).

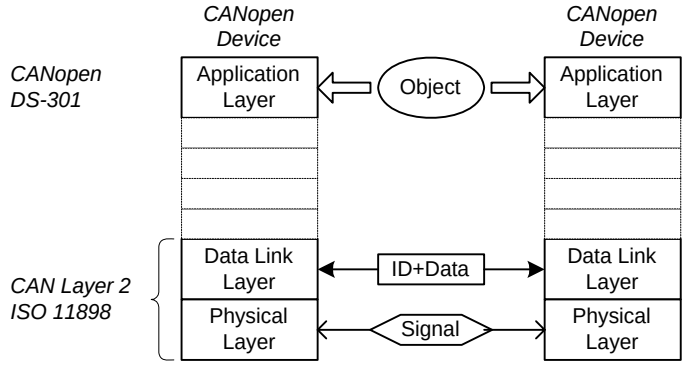


Figure 3-1: CANopen reference layer model

The CANopen Network can cover up to 128 devices, 127 Slaves and one Master. Each device is uniquely identified by its **Node-ID**.

The devices exchange data via **communication objects** of certain types which are handled by standardized **communication services**.

The communication objects are described within the Object Dictionary and mapped to a certain identifier (COB-ID). This mapping is also distributed within the Object Dictionary. It is pre-defined by the **Pre-Defined Connection Set** in [CiA 301] and can be remapped by the customer application.

Within the CANopen network a Master device fulfills all Network Management (NMT) tasks for controlling and monitoring of the Slaves. Additionally, the Master can comprise an SDO Manager and a Configuration Manager to realize SDO connection handling and Slave configuration.

3.1.2 CANopen Device

In CANopen devices three components cooperate to realize the specified device functionality (see Figure 3-2).

The **process interface** implements the overall functionality of the device. It interacts with the process by variables referred as application objects.

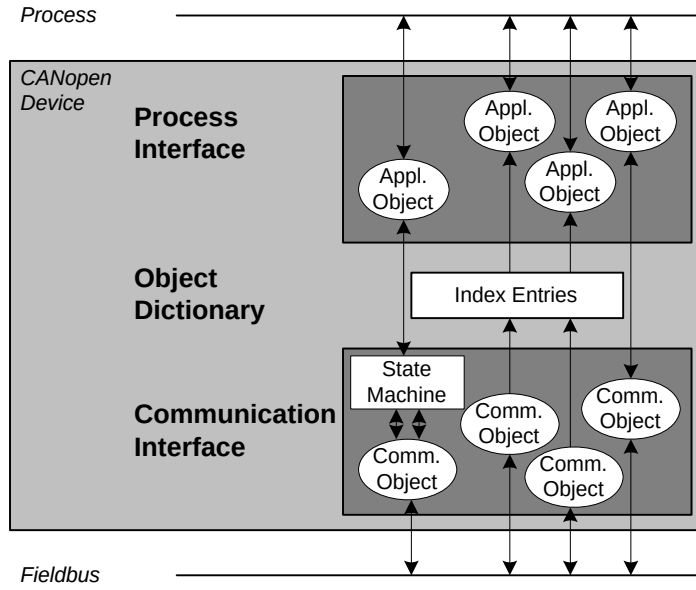


Figure 3-2: CANopen device model

As a central part of a CANopen device the **Object Dictionary** describes the used data types and communication objects. It also maps the application objects to the communication objects and their related identifiers (COB-ID).

The communication objects are transferred via the CANopen Network by the **communication interface**. The state machine of the CANopen device determines the communication behavior and is shown in Figure 3-3. All involved states and state transitions are described shortly in Table 3-1.

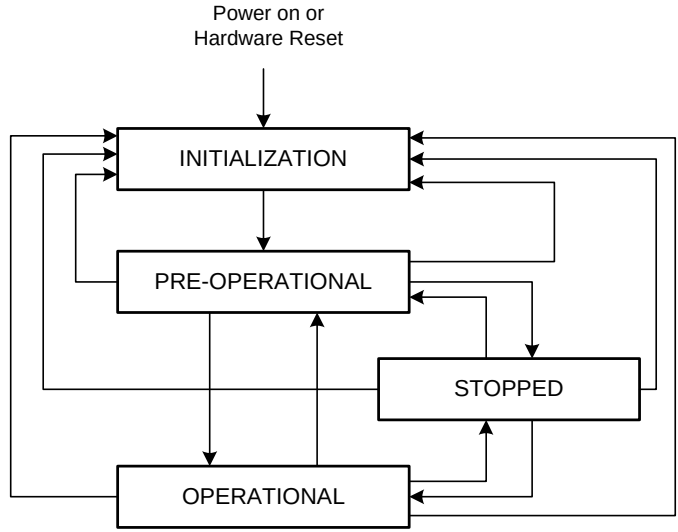


Figure 3-3: CANopen state machine

Table 3-1: CANopen device states

State	Remarks
PRE-OPERATIONAL	SDO Transfer is possible allowing parameter configuration as well as PDO configuration and mapping. Node guarding and SYNC processing is possible. PDO transfer is not allowed. On reception of Start Remote Node request the node is switched to OPERATIONAL.
OPERATIONAL	All communication objects can be transferred. All services can be performed.
STOPPED	Communication is stopped except Error Control Services. Also called “PREPARED” in CiA 301 V3.0
Further intermediate states that finally result in state PRE-OPERATIONAL	
RESET APPLICATION	Standard and manufacturer-specific parameters are set to their default values. Autonomous state change to RESET COMMUNICATION.
RESET COMMUNICATION	Communication parameters are set to the default values. Autonomous state change to INITIALIZING.
INITIALIZING	Basic node initialization. Transmission of the boot-up object. Autonomous state change to PRE-OPERATIONAL.

3.1.3 CANopen Communication Objects and Services

CANopen uses several communication object types for data exchange, network administration and device configuration. The communication objects are handled by several transfer services with specified protocols (see Table 3-2).

Table 3-3 lists active communication objects depending on the CANopen device states. The objects and their services are shortly described in the following sections.

Table 3-2: Communication objects and services

Object	Service
Process Data Objects (PDO)	Write PDO Remote PDO Request
Service Data Objects (SDO)	Initiate SDO Download Download SDO Segment Initiate SDO Upload Upload SDO Segment Abort SDO Transfer
Synchronization Object (SYNC)	SYNC transmission
Network Management Objects (NMT)	Module Control Services (State change) Error Control Services (Node and life guarding)
Emergency Object (EMCY)	EMCY transmission
Time Stamp Object (TIME)	TIME transmission
Boot-up Object	Boot-up service

Table 3-3: Active communication objects

	INITIALIZING	PRE-OPERATIONAL
SDO		X
PDO		
SYNC		X
NMT		X
Boot-up	X	
EMCY		X
TIME		X
	OPERATIONAL	**STOPPED**
SDO	X	
PDO	X	
SYNC	X	
NMT	X	X
Boot-up		
EMCY	X	
TIME	X	

x: available

3.2 Physical Layer

CANopen devices are connected via a two-wire bus line according to the CAN High-Speed Specification (ISO 11898-2). The bus lines are terminated by 120 Ω resistors at both ends (see Figure 3-4).

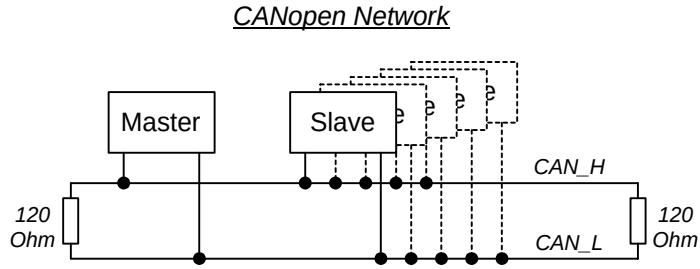


Figure 3-4: CANopen network

The CANopen specification recommends certain baudrates and sample points (see Table 3-4). At least one of the listed baudrates has to be applied to the network.

The maximum bus length depends on the baudrate and the applied cable type. For a bus length greater than 1000 m repeater devices may be employed.

Table 3-4: CANopen bit rates

Bit rate [kbit/s]	Bit time [µs]	Sample point (1) [µs]	Bus length (2) [m]
1000	1	0,75	25
800	1,25	1	50
500	2	1,75	100
250	4	3,5	250
125	8	7	500
50	20	17,5	1000
20	50	43,75	2500
10	100	87,5	5000

(1) Recommendation
(2) Worst case

3.3 Process Data Object (PDO)

PDOs are used to transfer real-time application data. They can include a maximum of 8 data bytes. Number, length and represented application data of the PDOs are specified in the device profile and mapped together with the COB-ID in the Object Dictionary.

The objects are assigned a certain type out of a choice of different transfer behaviors. All PDO transfers are managed by type dependant unconfirmed services which base on the producer/consumer (broadcast) model.

3.3.1 PDO Types

Basically, the PDOs can be distinguished by their **communication direction**

- Receive PDO (RPDO)
- Transmit PDO (TPDO)

transmission mode

- Synchronous (cyclic/acyclic)
- Asynchronous

and **triggering mode**

- Event triggered (Timer, SYNC object, internal event)
- Remotely requested (RTR)

The combinations of these transfer characteristics result in certain PDO transmission types which are assigned by a standardized type number (see Table 3-5).

Receive PDO (RPDO)

- Synchronous RPDO (0-240)
  The received PDO is passed to the consuming application at the reception the next SYNC object (see section 3.5).
- Asynchronous RPDO (254, 255)
  The received PDO is passed to the application immediately.

Transmit PDO (TPDO)

- Acyclic synchronous TPDO (0)
  The PDO is sent within the synchronous time window after occurrence of the SYNC object (see section 3.5) in case of a prevailing application-specific event.
- Cyclic synchronous TPDO (1-240)
  The PDO is sent within the synchronous time window after occurrence of the n-th SYNC where n is equally to the transmission type number (see Table 3-5).
- Remotely requested synchronous TPDO (252)
  The PDO is sent within the synchronous time window after occurrence of the SYNC object in case of a prevailing remote request by another device.
- Remotely requested asynchronous TPDO (253)
  The PDO is transmitted immediately after reception of a remote request by another device.
- Asynchronous TPDO (254, 255)
  The PDO is sent immediately on occurrence of an application-specific event.

Table 3-5: PDO transmission types

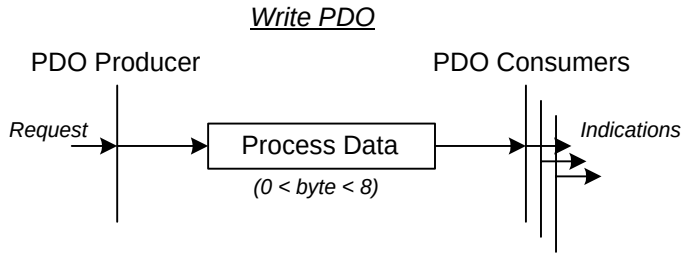
Transmission Type Number	Cyclic	Acyclic	Sync.	Async.	RTR only
0		x	x		
1-240	x		x		
241-251	reserved				
252			x		x
253				x	x
254				x	
255				x	

3.3.2 PDO Services

The PDO transfer services require the CANopen devices to be in OPERATIONAL state. Depending on their transmission type the PDOs are transferred by following services:

- Write PDO
  (on application-specific event or reception of a SYNC)
- Remote PDO Request
  (on request of a consumer)

The service protocols base on the Producer/Consumer model and are shown in Figure 3-5.



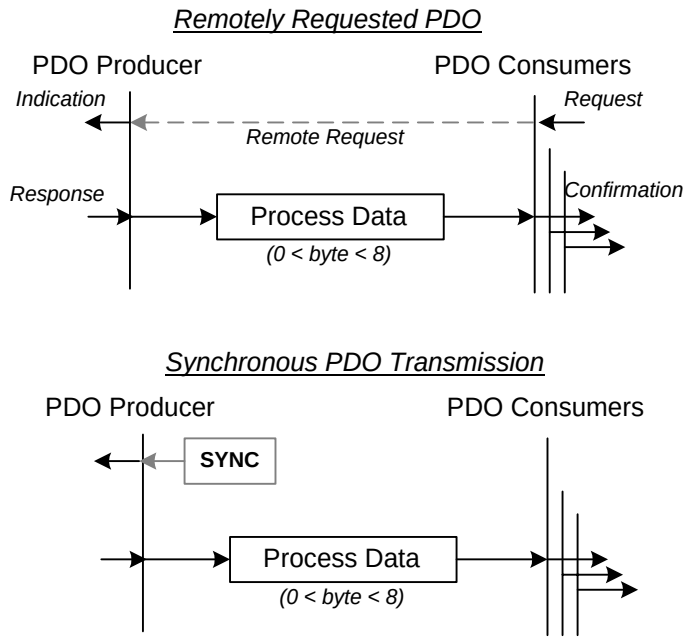


Figure 3-5: PDO transmission service protocols

3.4 Service Data Object (SDO)

SDOs provide access to the Object Dictionary of a CANopen device. An SDO can bear multiple data sets of arbitrary type and size. Thus, it can be used to transfer data of any size as well as configuration data. The data transfer bases on a Client/Server relationship.

The server is the owner of the accessed object dictionary. Initiating the SDO transfer the client determines index and sub-index of the data set to be transferred. The SDO access is realized by confirmed services (see Figure 3-6 and Figure 3-7).

3.4.1 SDO Download

Two services write the SDO data to the Object Dictionary of the server:

Initiate Download

Initiating the SDO download the client indicates length, index and sub-index of the data set. The initiating service message is confirmed by the server.

In case of an **expedited download** (SDO data length ≤ 4 bytes) the SDO data are included in the initiating service. Occurred transfer errors are signalled within the confirmation of the server.

Download Segment

If the SDO data size is larger than 4 bytes the data set is divided in segments of maximum 7 bytes which are successively downloaded. Transfer failures and error causes are provided within the confirmation of the server.

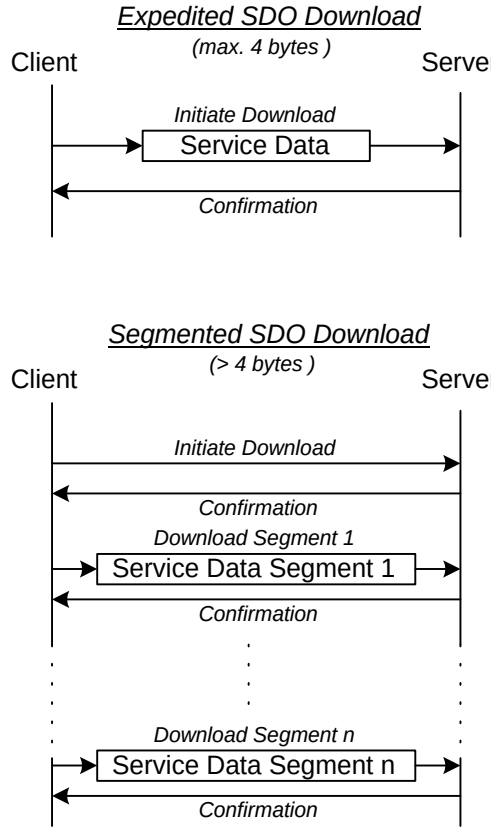


Figure 3-6: SDO download services

3.4.2 SDO Upload

Reading data from the Object Dictionary of the server is processed by two services:

Initiate Upload

Initiating the SDO upload the client indicates length, index and sub-index of the requested data set. The initiating service message is confirmed by the server.

In case of an **expedited upload** (SDO data length ≤ 4 bytes) the SDO data are included in the service confirmation of the server. Occurred transfer errors are also signalled within the confirmation.

Upload Segment

If the data size is larger than 4 bytes the data set is divided in segments of maximum 7 bytes. The client sends the upload segment requests to the server. The confirmation of that requests includes the related data segment. Transfer failures and error causes are also provided within the confirmation of the server.

3.4.3 Abort SDO Transfer

SDO Uploads and Downloads can be aborted at any time by the client or the server. The related abort service is unconfirmed and may include the abort reason.

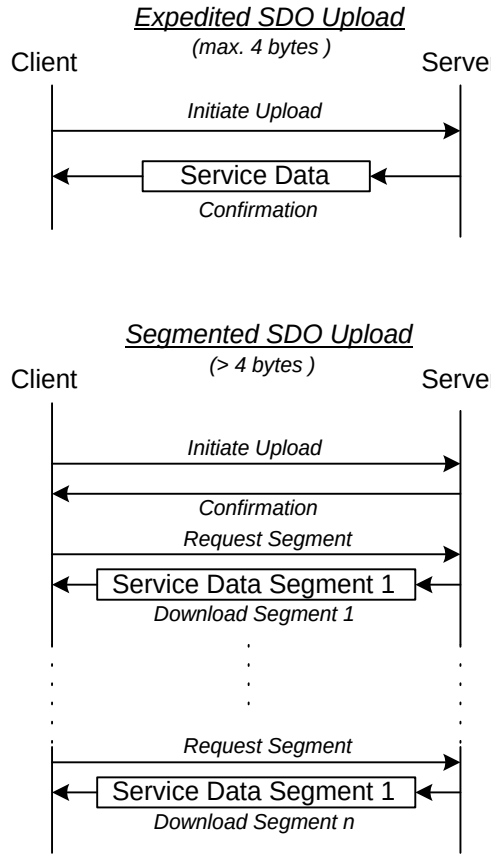


Figure 3-7: SDO upload services

3.4.4 SDO Block Transfer

SDO Block Transfers are variations of the SDO Transfers that are optimized for throughput. While SDO download and SDO upload require a request and a confirmation for each transferred data segment the SDO Block Transfers can handle multiple data segments being transferred without the receiving side answering to each segment. The SDO Block Transfers allow for a configurable number of data segments (1 … 127) that are secured by a single handshake, thus reducing the required bandwidth and latency especially for large data blocks.

3.5 Synchronization (SYNC Object)

Automation systems often require a synchronous behavior of certain components. In CANopen networks the synchronization is realized by a specific communication object (SYNC).

The SYNC producer broadcasts the SYNC message periodically on the network while the SYNC consumers execute the synchronous tasks on reception of the SYNC object.

The synchronization tasks are device dependant. Beside the time fixed transmission of the synchronous TPDOs (see Figure 3-8) the synchronization involves the process actuation and data sampling.

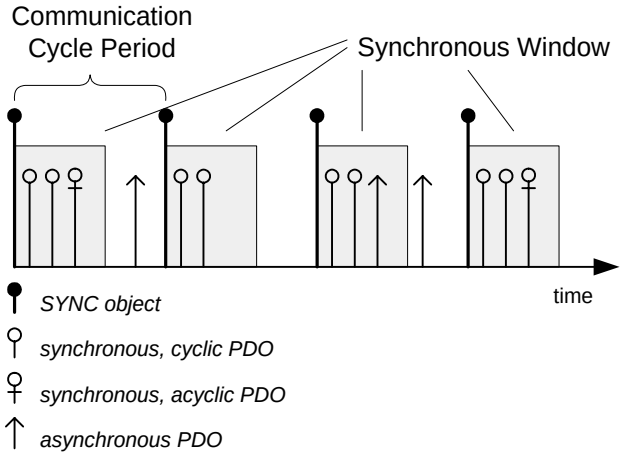


Figure 3-8: Synchronous communication

3.6 Network Management (NMT)

The Network Management (NMT) services are based on a Master/Slave relationship. They comprise services from network initialization, error control and device state control.

The NMT services are executed by the NMT Master using certain NMT objects.

3.6.1 Module Control Services

The NMT Master controls the states of the Slaves by the Module Control Services. These services are unconfirmed. They can be performed for a single Slave, the Master or all present devices simultaneously.

The Master transmits the NMT object (COB-ID=0) determining the target device and the state transition which should be performed.

Table 3-6 enumerates the available services and the related states to which the selected device is switched (see also section 3.1).

Table 3-6: Module Control Services

Service	Target State
Start Remote Node	OPERATIONAL
Stop Remote Node	STOPPED
Enter Pre-Operational	PRE-OPERATIONAL
Reset Node	RESET APPLICATION
Reset Communication	RESET COMMUNICATION

3.6.2 Error Control Services

There are two different methods to determine errors in a network - Node/Life Guarding and Heartbeat. Both allow for supervision of Master and Slaves. The two methods can not be used in parallel. Using one of these methods is mandatory.

Node/Life Guarding

The NMT Master polls the Slaves cyclically by a remote transmit request (COB-ID = 1792+Node-ID). The cyclic period of the guard request (**Node Guard Time**) is individually defined for each Slave. A guarded Slave answers to the remote request with its current state.

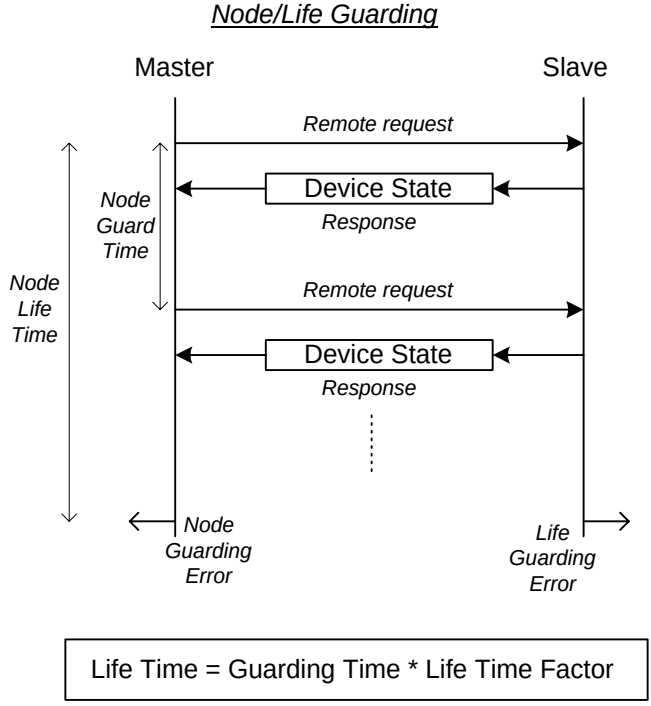


Figure 3-9: Node/Life guarding protocol

A **remote node error** is signalled by the NMT Master if

- the Slave doesn't answer within the defined **Node Life Time** or
- the answer of the addressed Slave reports an unexpected state.

If Live Guarding is supported the Slave observes the guarding of the NMT Master. In this case the Slave reports a Life Guarding Error to its application if it is not guarded within its Life Time.

Heartbeat

A Heartbeat producer cyclically transmits a Heartbeat message (COB-ID = 1792+Node-ID) that contains its current state. The cyclic period of the Heartbeat (**Heartbeat Producer Time**) is individually defined for each device.

Any Heartbeat consumer can observe the occurrence of a device's Heartbeat message by an individually defined **Heartbeat Consumer Time.** If the Heartbeat message does not occur within this time a Heartbeat error is signalled.

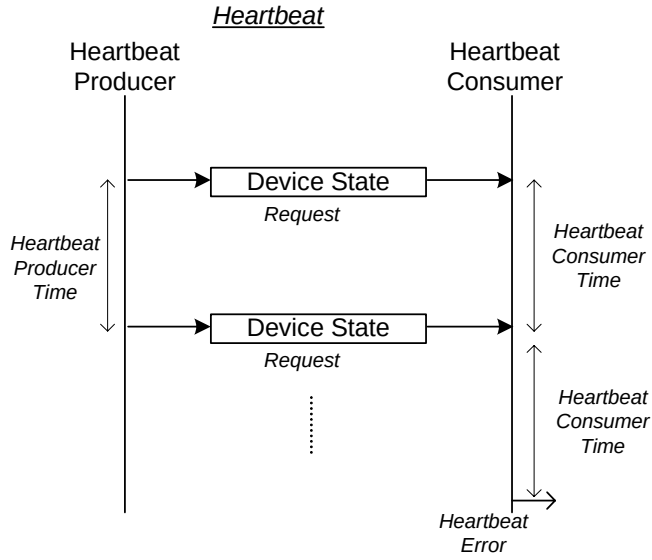


Figure 3-10: Heartbeat protocol

3.6.3 Boot-up service

Entering the PRE-OPERATIONAL state from INITIALIZATION the CANopen devices introduce themselves to the network transmitting a boot-up object (COB-ID = 1792+ Node-ID).

3.6.4 Emergency Object (EMCY)

The optional Emergency Object (COB-ID = 128 + Node-ID) is transmitted by a CANopen device in case of an internal error event or situation. It includes an error code number which refers to the error type or cause.

3.7 Object Dictionary

The Object Dictionary is the central part of a device profile. It lists all objects, their data types and attributes which are available via the CANopen network.

The entries of the Object Dictionary are accessed by their index. If the related object is a data record or array a sub-index references the contents of the structured object.

The available objects and their index and sub-index are available in the certain device profiles [CiA 40x].

4 CANopen Client API

4.1 Common

4.1.1 CANopen Client API Concept

The CANopen Client API is compatible with the CANopen specification [CiA 301 V 4.0.2]. It is implemented as a 32bit Windows DLL that makes use of Softing's speed-optimized CAN Layer2 API.

It comprises the following components:

Layer Manager
Initializes and observes the physical layer.

SDO Manager
for SDO download and upload services including an SDO transfer observation by timeout.

Node Manager
for administration of the Slave nodes and realization of the NMT Master services

- Module State Control
- Emergency (EMCY) support
- Error Control (Heartbeat, Node Guarding)
- Boot-Up Service

SYNC Manager
producing and/or consuming SYNC messages.
It manages every task related to the synchronous behaviour of the CANopen Client.

PDO Manager
for administration of the PDO buffer and realization of the PDO transmission services

PDO Buffer
holds a consistent copy of the actual process data base. It can bear a maximum of 512 TPDOs and 512 RPDOs.

Event-FIFO

Events can occur asynchronously. The *Event-FIFO* can buffer up to 255 events. Its handling is described in section 4.7.

4.1.2 Driver Concept

All functions of the CANopen Client API are supplied in the Windows DLL *CANopenL2.dll*.

This is a 32bit DLL that can be used by 32bit application programs running either on 32bit or 64bit Windows operating systems. All CANopen specific functionality is included in this DLL.

The *CANopenL2.dll* itself calls the underlying CAN Layer2 API DLL that offers CAN bus access on message level. Please see the “Softing CAN Layer2 Manual” for details of the driver architecture.

4.1.3 Main Programming Sequence

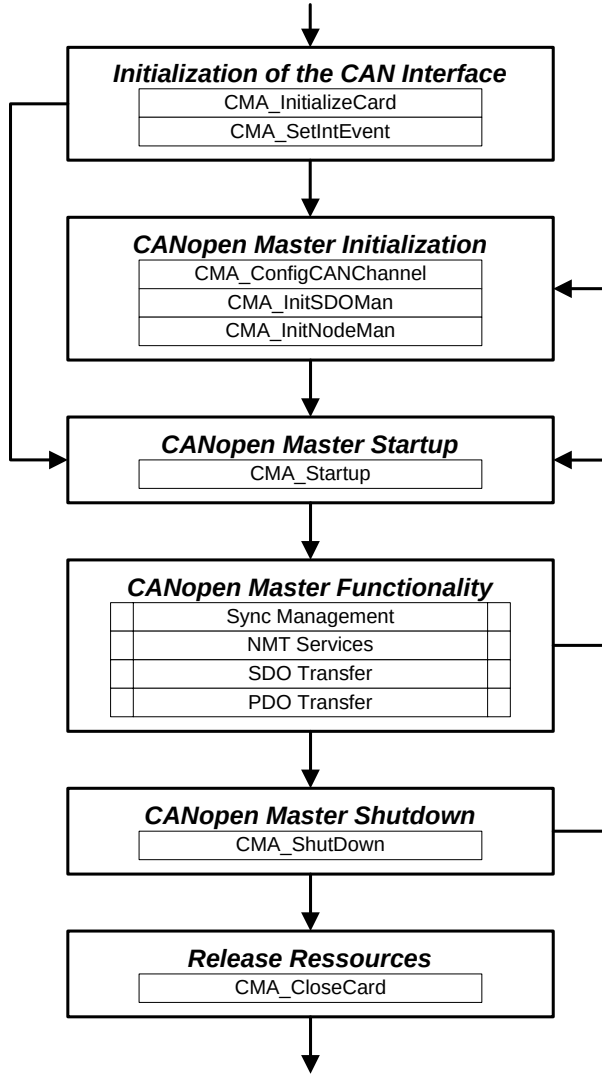


Figure 4-1: Main programming flow chart

4.2 Initialization

4.2.1 Initialization of the CAN Channel

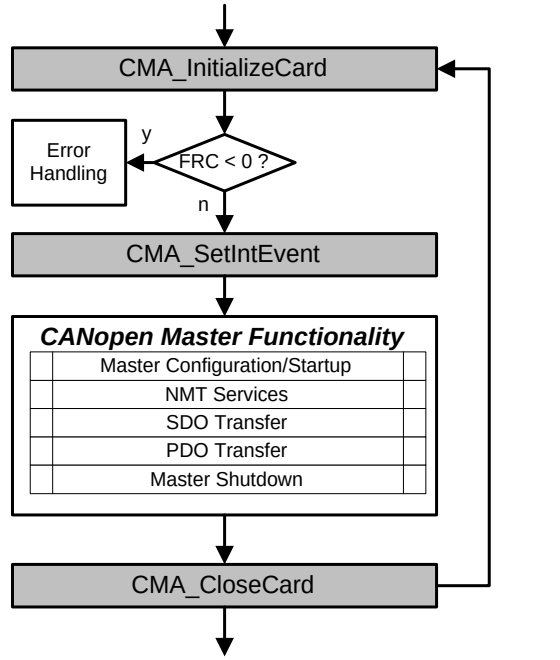
Implementation

1. Before using the CAN channel as a CANopen Client the required hardware and **system resources** have to be allocated and assigned to the application. This task is performed calling *CMA_InitializeCard* (see section 6.8).
2. In case of a CANopen event the driver informs the application by signalling a WIN32 event. As a prerequisite the handle of this WIN32 event needs to be assigned to the driver by *CMA_SetIntEvent*.
3. If the CANopen Client application is closed after a successful initialization the locked resources are to be released by *CMA_CloseCard*.

Prerequisites

- Proper driver installation (see Chapter 2).
- Valid WIN32 event handle (only if interrupt is used).

Sequence



s

Figure 4-2: Initialize interface flow chart

Programming Notes

The function return code (FRC) of *CMA_InitializeCard* should be evaluated to prevent access errors of succeeding function calls. A prepared evaluation routine is available in the source examples.

CMA_SetIntEvent may be called at any time after *CMA_InitializeCard* but before the first occurrence of an interrupt (e.g. performing start-up).

CMA_SetIntEvent can be omitted if the application does not use the interrupt.

Assure that *CMA_CloseCard* is performed at any possible exit of the CANopen Client application.

Omitting the release of the allocated resources by *CMA_CloseCard* at program exit may cause problems calling *CMA_InitializeCard* again.

Sample Programs and Source Code

Interrupt.exe
Interrupt.c

4.2.2 Hardware and Software Version Information

Implementation

In some applications the versions of the applied software and hardware must be evaluated for error protection or customer information reasons. These data can be accessed by calling *CMA_GetVersion*.

Prerequisites

- Proper driver installation (see Chapter 2).
- Valid WIN32 event handle (only if interrupt is used).

4.2.3 Start-up and Shutdown of the CANopen Client

Implementation

After initialization of the CAN channel the CANopen Client is started up calling *CMA_Startup.*

Activities of *CMA_Startup*:

1. Initialization of the Master components using the parameters listed in Table 4-1.
2. Reset, initialization and start of the CAN controllers.
3. Transmission of the boot-up message (COB-ID: 1792+Node-ID, default: 1919).

Table 4-1: CANopen Client initialization parameters

Master Components	Initialization Parameter	Pre-setting (default values)
Layer 2 Management	Baudrate	125 kbit/s
Node Management	Client Node-ID	127
SDO Management	SDO Timeout	2 s
SDO Management	Timeout Factor	1

The **initialization parameters** are preset to default values. They can be adjusted at customer's choice by the API functions in Table 4-2.

Table 4-2: CANopen Client initialization functions

Initialization Parameter	Adjustable by
Baudrate	*CMA_ConfigCANChannel*
Client Node-ID	*CMA_InitNodeMan*
SDO Timeout	*CMA_InitSDOMan*
SDO Timeout Factor	*CMA_InitSDOMan*

After start-up the CANopen Client is in **PRE-OPERATIONAL** state, i.e.:

- Client is active on the network
- SDO transfer is possible.
- Heartbeat or node guarding can be started.
- SYNC production/consumption can be started.
- No PDO transfer.

If the CANopen Client is in PRE-OPERATIONAL or OPERATIONAL state it is also termed to be **'active'**.

Complementary to the start-up the **shutdown** of the CANopen Client is performed by *CMA_Shutdown*, i.e.:

- CAN controller is reset.
- CANopen Client is logically removed from the network.
- Parameters in Table 4-1 are reset to their defaults.

The Client can be **restarted without a complete shutdown** at any time calling *CMA_Startup* again. The customized settings of the parameters in Table 4-1 are preserved in this case. Any other Client configurations and settings are reset to their default values or cleared, i.e.:

- Heartbeat / Node guarding is stopped.
- SYNC handling is stopped.
- PDO buffer is cleared.
- Client is reinitialized and switched to PRE-OPERATIONAL.
- Client is active on the bus

Prerequisites

- A valid CAN handle provided by CMA_*InitializeCard.*

Sequence

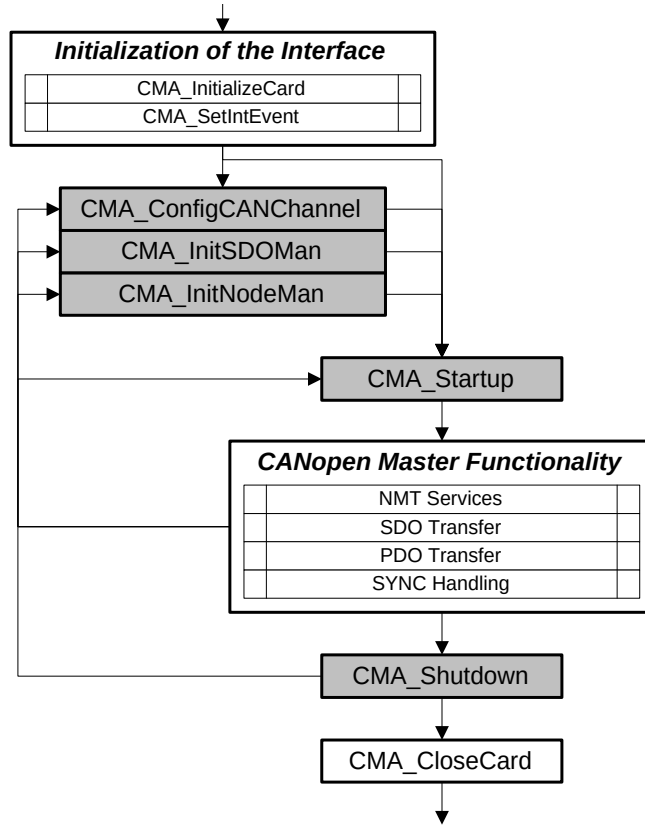


Figure 4-3: CANopen Client start-up/shutdown flowchart

NOTE:
The start-up parameters can also be adjusted while the Client is active. Anyway, the new settings are applied during the next start-up.

Programming Notes

- The optional configuration of the start-up parameters by

  CMA_ConfigCANChannel
  CMA_InitNodeMan
  CMA_InitSDOMan

  can be performed at any time after the initialization of the CAN channel. The values are buffered and are taken into effect during the next start-up. If the parameter configuration is omitted, the Client starts up using the default values (see Table 4-1).

- If you restart the CANopen Client the node list of the node manager is cleared and needs to be reinitialized (*CMA_AddNode*).

- If you restart the CANopen Client a previous Heartbeat or Node Guarding is stopped and needs to be restarted (*CMA_RequestGuarding*).

- If you restart the CANopen Client a previous SYNC producing or consuming is stopped and needs to be reconfigured (*CMA_ConfigSyncMan*).

Sample Programs and Source Code

All sample programs and source code.

4.3 SDO Transfer

4.3.1 Common

The CANopen Client uses the default SDOs of the Predefined Connection Set as specified in [CiA 301], i.e. one SDO per node.

The target of the SDO transfer is the Object Dictionary either of the CANopen Client or a connected Slave.

Timeout

The SDO connections between the CANopen Client and the Slave nodes are observed by a timeout. This timeout is configured by parameter *usSDOTimeout* during start-up (see section 4.2.3).

The timeout starts when the Client is transmitting its SDO transfer request to the slave (see section 3.4). If the timeout expires without a response or confirmation of the Slave an *SDOError* event is signalled to the application.

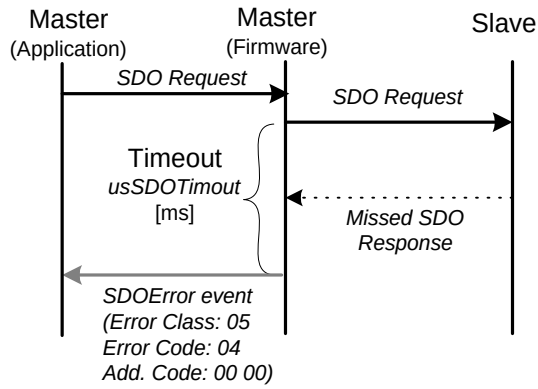


Figure 4-4: SDO transfer timeout

4.3.2 SDO Download

Implementation

Initiate the download by *CMA_InitDownload or by CMA_InitBlockDownload* supplying

- Node-ID of the target
- SDO index and sub-index
- Overall data length
- SDO data

For *CMA_InitDownload t*he CANopen Client API evaluates the data length and decides autonomously if the data fit into an expedited SDO download or if a segmented SDO Download is required. Successful SDO downloads, regardless of their type (expedited, segmented or block) are confirmed by the event *DownloadCompleted* in the *Event-FIFO*. Occurred transfer errors are passed to the application by the *SDOError* event including the related SDO error class, error code and additional code (see section 4.7).

Prerequisites

1. A valid CAN handle provided by CMA_*InitializeCard.*
2. CANopen Client is active and in OPERATIONAL or PRE-OPERATIONAL state.

Sequence

Fehler! Es ist nicht möglich, durch die Bearbeitung von Feldfunktionen Objekte zu erstellen.

Figure 4-5: SDO download flow chart

Programming Notes

If the Slave aborts the SDO transfer the Client application is informed by a related *SDOError* event in the *Event-FIFO*.

The Object Dictionary entries of the CANopen Client (see section 4.7) can be written by SDO download referring to the Client Node-ID. The Client Node-ID is configured by *CMA_InitNodeMan*.

All download commands are operated asynchronously. The functions don't return any confirmations of the actual SDO transfer. Such transfer confirmations or errors are signalled by following events via the *Event-FIFO* (see section 4.7):

DownloadCompleted
SDOError

Sample Programs and Source Code

SDODownload.exe
SDODownload.c

4.3.3 SDO Upload

Implementation

1. Initiate the download by *CMA_InitUpload* supplying

   - Node-ID of the source
   - SDO index and sub-index

   or by *CMA_InitBlockUpload* supplying

   - Node-ID of the source
   - SDO index and sub-index
   - Number of segments within one block (1 …127)

2. Wait for the *UploadCompleted* event in the *Event-FIFO*. It contains the uploaded SDO data and data length. In case of an expedited upload the event also contains the uploaded data.
3. For segmented and block uploads the uploaded data need to be read by *CMA_ReadData.*

Successful SDO uploads, regardless of their type (expedited, segmented or block) are confirmed by the event *UploadCompleted* in the *Event-FIFO*. Occurred transfer errors are passed to the application by the *SDOError* event including the related SDO error class, error code and additional code (see section 4.7).

Prerequisites

1. A valid CAN handle provided by CMA_*InitializeCard.*
2. CANopen Client is active and in OPERATIONAL or PRE-OPERATIONAL state.

Sequence

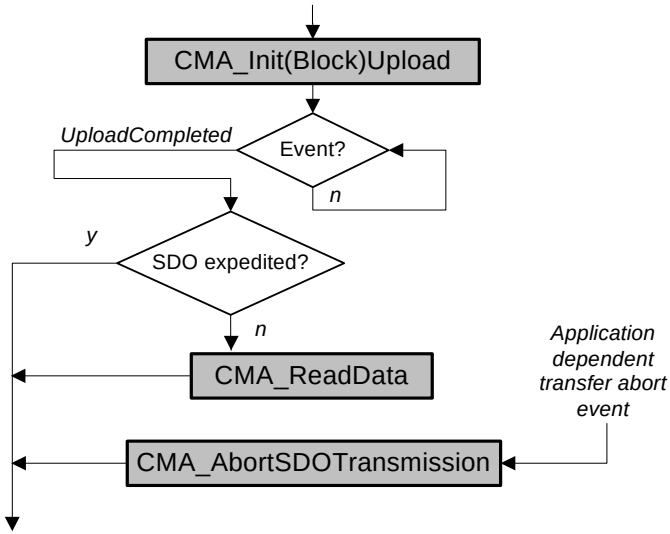


Figure 4-6: SDO upload flow chart

Programming Notes

If the Slave aborts the SDO transfer the Client application is informed by a related *SDOError* event in the *Event-FIFO*.

The Object Dictionary entries of the CANopen Client (see section 4.7) can be read by the SDO upload referencing the Client Node-ID in the function parameter. The Client Node-ID is configured by *CMA_InitNodeMan*.

All upload commands are operated asynchronously. The functions don't return any confirmations of the actual SDO transfer. Such transfer confirmations or errors are signalled by following events via the *Event-FIFO* (see section 4.7):
UploadCompleted
SDOError

Sample Programs and Source Code

SDOUpload.exe
SDOUpload.c

4.3.4 Abort SDO Transfer

A pending SDO transfer can be aborted at any time by *CMA_AbortSDOTransmission* (see section 6.1). The abort service is submitted to the CANopen Client API asynchronously without any confirmation.

If the SDO transfer is aborted externally by the Slave or internally by the CANopen Client the application is informed by an error event in the *Event-FIFO* (see section 4.7). The event of *SDOError* event type includes the error cause encoded in error class, code and additional code as defined in the CANopen specification [CiA 301] (see Appendix A).

4.4 PDO Transfer

The CANopen Client buffers PDOs internally. This buffer can include up to 512 TPDOs and 512 RPDOs. Before the PDOs can be transmitted or received they have to be installed defining the PDO type and attributes.

4.4.1 PDO Buffer Administration

Implementation

Install PDO

TPDOs and RPDOs are installed by *CMA_InstallPDO_E* determining

- COB-ID
- Transmission type (see Table 6-1)
- Data length of the PDO
- Initial PDO data and start-up behaviour

The function returns a handle to the PDO which is used by other API functions to access the installed PDO.

For RPDOs the customer can define if the application is informed by a *PDOReceived* event in case of the reception of the PDO.

If the CANopen Client is switched from PRE-OPERATIONAL to OPERATIONAL state (see section 4.6.3) optionally all installed TPDOs may be transmitted once to the network automatically.

Read PDO Data

The actual data of the installed PDOs can be read from the PDO buffer by *CMA_ReadPDO.*

Remove PDO

The installed PDOs can be removed from the PDO buffer individually by *CMA_RemovePDO*. A shutdown or a repeated start-up of the CANopen Client removes the installed PDOs globally.

Prerequisites

1. A valid CAN handle provided by CMA_*InitializeCard.*
2. CANopen Client is active and in OPERATIONAL or PRE-OPERATIONAL state.
3. *CMA_RemovePDO and CMA_ReadPDO* require a valid PDO handle returned by *CMA_InstallPDO_E*.

Programming Notes

The configuration functions are commanded via the *Command Interface* and return the success or errors accessing the PDO buffer.

Overall, a maximum of 512 TPDOs and 512 RPDOs can be installed.

Sample Programs and Source Code

PDOTransfer.exe
PDOTransfer.c

4.4.2 PDO Services

Implementation

Transmit PDO

The data of the TPDOs are written to the PDO buffer by

- *CMA_WritePDO* refreshing all data bytes
- *CMA_WritePDOBit* refreshing a certain data bit

The CANopen Client API transmits the PDOs in the PDO buffer depending on the PDO transmission type chosen by *CMA_InstallPDO_E*. In Table 4-3 the instants of transmission are listed together with the available PDO transmission types.

Table 4-3: PDO transmission types and instants

PDO Transmission Type	Type Number	Transmission Instant
Acyclic synchronous	0	Next occurrence of the SYNC after *CMA_WritePDO* or *CMA_WritePDOBit*
Cyclic synchronous	1-240	Every n^{th} occurrence of the SYNC (n = PDO Transmission Type Number)
Synchronous (RTR only)	252	Next occurrence of the SYNC after reception of a remote frame of the same COB-ID.
Asynchronous (RTR only)	253	Reception of a remote frame of the same COB-ID.
Asynchronous	254, 255	After refresh of PDO data by *CMA_WritePDO* or *CMA_WritePDOBit*

Beside the transmission instants in Table 4-3 all installed PDOs are immediatly transmitted once on

- CANopen Client state change to OPERATIONAL (*CMA_ChangeState*) if initial transmission is enabled
- Reception of a remote frame of the same COB-ID (except TPDOs of type 252)

Remote PDO Request

PDOs provided by a Slave can be remotely requested by *CMA_SendRemotePDO*. Since the Slave replies by sending the related PDO data, a valid RPDO installation (*CMA_InstallPDO_E*) with the same COB-ID is required for proper operation. The application is informed about the reception of the replyed PDO by the event *PDOReceived* in the *Event-FIFO*.

Receive PDO (RPDO)

If an installed RPDO is received the CANopen Client API writes the data to the PDO buffer. The application is informed about the reception by the event *PDOReceived* in the *Event-FIFO* if the PDO was installed determining *ucEventNotif* $\neq 0$.

Prerequisites

1. A valid CAN handle provided by CMA_*InitializeCard.*
2. CANopen Client is active and in OPERATIONAL state.
3. A valid PDO handle returned by *CMA_InstallPDO_E*.

Programming Notes

PDOs in the PDO buffer are accessed referencing the PDO handle returned by *CMA_InstallPDO_E*.

All PDOs are globally removed by a Client shutdown (*CMA_Shutdown*) or repeated start-up (*CMA_Startup*).

The PDOs can be installed as soon as the CANopen Client is active, i.e. after start-up. But the PDO transfer requires the Client to be in OPERATIONAL state.

Sample Programs and Source Code

PDOTransfer.exe
PDOTransfer.c

4.5 Synchronization

Implementation

The SYNC Manager coordinates every task concerning the SYNC object, i.e.

- Transmission of SYNC (Producer)
- Initiation of synchronous PDO transmissions (Consumer)

During start-up the SYNC Manager is initialized to either produce or consume the SYNC object.

Start SYNC Transmission

The transmission of the SYNC object is started by *CMA_ConfigSyncMan* with parameter *ucProducerMode* $\neq 0$. The COB-ID and the cycle period of the SYNC object can be adjusted at customer's choice.

Stop SYNC Transmission

The transmission of the SYNC object is stopped by *CMA_ConfigSyncMan* determining *ucProducerMode = 0*.

Enable Synchronous PDO Transmission

To enable the transmission of synchronous PDOs the SYNC Manager must be additionally configured to act as SYNC consumer, i.e. calling *CMA_ConfigSyncMan* with *ucConsumerMode* $\neq 0$.

Prerequisites

1. CANopen Client is active and in OPERATIONAL or PRE-OPERATIONAL state.

Programming Notes

The SYNC Manager can be defined as Producer and Consumer of the SYNC object simultaneously.

The cycle period can be adjusted within the range 0...32 767 000 µs.

If the Client is shutdown (*CMA_Shutdown)* or restarted (*CMA_Startup*) the SYNC Manager is reset to the default parameters, i.e. no consumption or production of the SYNC object.

If synchronous PDOs are installed the SYNC Manager must be configured to consume the SYNC.

The transmission of the SYNC starts as soon as *CMA_ConfigSyncMan* returns successfully. The transmission of the synchronous PDOs starts when the CANopen Client is switched to OPERATIONAL state and the SYNC Manager is defined as Consumer.

Sample Programs and Source Code

PDOTransfer.exe

PDOTransfer.c

4.6 NMT Services

The CANopen Client API supports the NMT functionality as defined in [CiA 301], i.e. administration and observation of the Slave nodes in the CANopen network.

4.6.1 Node Manager Configuration

Implementation

Initialize the NMT Master

The Node Manager is initialized by configuring the Client Node-ID using *CMA_InitNodeMan*.

Register Slave Node

As a prerequisite for the NMT Master services the Slave nodes have to be registered by *CMA_AddNode* providing its Node-ID and guarding / heartbeat parameters.

Remove Slave Node

The registered Slave nodes can be individually removed from the node list by *CMA_RemoveNode*. In this case the guarding / heartbeat supervision of the Slave is stopped. Additionally, the Slave is switched into a defined state which is selected by the function parameter *ucState*.

Globally, the node list of the Node Manager is cleared by a restart (*CMA_Startup*) or a shutdown (*CMA_Shutdown*) of the CANopen Client.

Prerequisites

CANopen Client is active, i.e. in OPERATIONAL, PREPARED or PRE-OPERATIONAL state.

Programming Notes

The NMT Master services Node Guarding, Heartbeat supervision, Module State Control and EMCY Object processing are only performed for nodes which are added to the node list of the Node Manager.

The CANopen Client does not need to be added to the node list. Its Node-ID is configured by *CMA_InitNodeMan.* The states of the CANopen Client can be switched as soon as it is active (start-up).

Sample Programs and Source Code

NMTServices.exe
NMTServices.c

4.6.2 Node Guarding / Heartbeat Supervision

Implementation

Configuration of the Supervision Parameters

The parameters of the Node Guarding or Heartbeat supervision are defined registering the Slave to the node list of the Node Manager via *CMA_AddNode* (see section 4.6.2 and 6.2). The parameters cannot be reconfigured while the Slave is registered on the node list. If the parameters need to be changed the Slave has to be removed by *CMA_RemoveNode* and subsequently re-registered by *CMA_AddNode*.

Start Supervision

The Node Guarding or Heartbeat supervision of a Slave node is started by calling *CMA_RequestGuarding* with parameter *ucReqGuard* $\neq 0$.

Stop Guarding

The Node Guarding of a Slave node by the CANopen Client is stopped calling *CMA_RequestGuarding* with parameter *ucReqGuard = 0*.

The guarding is also stopped removing the Slave node via *CMA_RemoveNode*.

Guarding Errors

Detected guarding errors or problems are posted to the application by the event *ErrorEvent* of type *GuardError* in the

Event-FIFO (see section 4.7). Table 4-4 lists the detected guarding errors and their error codes.

Table 4-4: Guarding errors

Error Code	Guarding Error
1	Guarding is active. A valid guard response was received after a preceding guard error 5.
2	No response of the Slave to a guard request within the guard time.
3	No response of the Slave to a guard request within the guard time and the retries (*ucRetryFactor*) are expired.
4	Toggle-Bit error in the guard response of the Slave.
5	Slave state has changed unexpectedly.

Prerequisites

1. CANopen Client is active, i.e. in OPERATIONAL, PREPARED or PRE-OPERATIONAL state.
2. Slave node is registered in the node list (*CMA_AddNode*).

Programming Notes

If the CANopen Client is restarted or shut down the guarding / heartbeat supervision of the Slaves is stopped globally.

Each Slave can be configured either for Node Guarding, or for Heartbeat supervision. A Slave can not use Node Guarding and Heartbeat supervision in parallel.

Sample Programs and Source Code

NMTServices.exe
NMTSevices.c

4.6.3 Module State Control

Implementation

The states of the registered Slaves and the CANopen Client can be switched by *CMA_ChangeState* defining the target state in the parameter *ucState* to

STOPPED
OPERATIONAL
RESET NODE
RESET COMMUNICATION
PRE-OPERATIONAL

The states are described in section 3.1.2 and 3.6.1.

The target node is addressed by its Node-ID. If the Client's state is to be switched the Client Node-ID or 0 must be defined in the parameter *ucNodeID*. Setting *ucNodeID* to 128 switches the states of all registered Slaves and the Client simultaneously.

Prerequisites

1. CANopen Client is active and in OPERATIONAL, PREPARED or PRE-OPERATIONAL state.
2. Slave nodes are registered in the node list (*CMA_AddNode*).

Programming Notes

After start-up the CANopen Client is in PRE-OPERATIONAL state. To enable PDO transfer it needs to be switched to OPERATIONAL state by *CMA_ChangeState*.

Sample Programs and Source Code

NMTServices.exe
NMTSevices.c

4.6.4 EMCY Object

Received emergency messages (COB-ID = 128 + Node-ID) of the Slaves are posted to the application by the *EMCYReceived* event in the *Event-FIFO* (see section 4.7). The event includes the data bytes of the EMCY object which are defined in the [CiA 301].

4.6.5 Boot-Up Service

The CANopen Client automatically transmits its boot-up message (COB-ID = 1792 + Node-ID) during start-up (*CMA_Startup*).

Received boot-up messages of registered Slaves are detected and evaluated by the Node Guarding (see section 4.6.2)

4.7 API Events and Event-FIFO

4.7.1 Reading the Event-FIFO

The CANopen Client informs the application about certain events which are posted to the application via the *Event-FIFO*. Additionally, a WIN32 event is triggered by the API.

The posted event messages are read out of the *Event-FIFO* calling *CMA_ReadEvent* which may be implemented in an event thread (see section 0) or a polling routine. The returned parameter structure includes the event type and the data (see section 4.7.2). For the event *UploadCompleted* the additional parameter *ulDataLenSegmented* indicates the total number of bytes included in the respective SDO upload.

The *Event-FIFO* can hold up to 255 entries. Thus it enables the application to overcome short service breaks without losing event messages.

4.7.2 Event Types and Data

The occurred events are classified by their event type in ucEventType and related event data in the ucaEventData array (see Table 4-5 and

Table 4-6).

Table 4-5: Event types

Event Type (*ucEventType*)	Description
(0) *NoEvent*	*Event-FIFO* is empty.
(3) *DownloadCompleted*	SDO download completed successfully. (see section 4.3.2)
(5) *UploadCompleted*	SDO upload completed successfully. (see section 4.3.3)
(11) *EMCYReceived*	Emergency message of a Slave node received.
(13) *BootUpReceived*	Boot-up message of a Slave node received.
(15) *ErrorEvent*	API, CAN, SDO or Guarding error (see Table 4-7).
(30) *NextDownloadSegment*	Not used (for compatibility only)
(31) *NextUploadSegment*	Not used (for compatibility only)
(32) *PDOReceived*	An RPDO was received. (see section 4.4.2)
(33) *ShutDownCompleted*	CANopen Client shutdown completed successfully. (see section 4.2.3)
(34) *StartupCompleted*	CANopen Client start-up completed successfully. (see section 4.2.3)

Table 4-6: Event data

Event Type	Event Data Array (*ucaEventData)*	
	No.	Data
NoEvent		None.
DownloadCompleted	[0]	Node-ID of the SDO transfer target.
	[1]	SDO sub-index.
	[2,3]	SDO index (low, high byte)
UploadCompleted	[0]	Node-ID of the SDO transfer source.
	[1]	SDO sub-index.
	[2,3]	SDO index (low, high byte)
	[4]	SDO data length. FFH: Segmented SDO; no data included; data length in element *ulDataLenSegmented* others: expedited SDO; length of included data
	[5..]	Expedited upload data
EMCYReceived	[0]	Node-ID of the Slave.
	[1...8]	EMCY object data [0...7]
BootUpReceived	[0]	Node-ID of the Slave.
ErrorEvent	[1...]	Error type and data (see Table 4-7)
PDOReceived	[0,1]	RPDO handle (low, high)
	[2]	Received data length.
	[3...]	Received data.
ShutDownCompleted		None.
StartupCompleted		None.

Table 4-7: Error event types and data

Type No. (*)	Error Event Type	Event Data Array (*ucaEventData*)	
		Field No.	Data
1	*API Error*	[1]	Reserved.
		[2,3]	Error code (low, high) 0x0051: *Event-FIFO* overrun. 0x0052: RPDO could not be written. 0x0053: TPDO could not be read. others: Internal errors Restart of the Client necessary.
2	*CAN Error*	[1]	Reserved
		[2,3]	Error code (low, high)
			0x0002: TX queue overrun. Transmit request is lost.
			0x0100: RX queue overrun.
			0x0101: CAN controller state changed to BUS OFF. Restart of the Client necessary.
			0x0102: CAN controller state changed to ERROR PASSIVE.
			0x0103: CAN controller state changed to ERROR ACTIVE.

(*) *ucEventData[0]*

Type No. (*)	Error Event Type	Event Data Array (*ucaEventData)*	
		Field No.	Data
3	*SDO Error*	[1]	Node-ID of the target or source
		[2]	SDO error class (see Appendix A)
		[3]	SDO error code (see Appendix A)
		[4,5]	Additional SDO error code (see Appendix A)
4	*Guard Error*	[1]	Node-ID of the Slave
		[2]	Error Code (see Table 4-4)

(*) *ucEventData[0]*

4.8 Client Object Dictionary

The local Object Dictionary of the CANopen Client API includes the entries listed in Table 4-8. They can be accessed using the SDO download and upload services described in section 4.3.

Table 4-8: Object Dictionary entries of the CANopen Client

Object	Index
Device Type	1000H
Error Register	1001H
SYNC COB-ID	1005H
Communication Cycle Period	1006H
Manufacturer Device Name	1008H
Manufacturer Hardware Version	1009H
Manufacturer Software Version	100AH
Identity Object	1018H

5 Programming Notes

5.1 API Function Linking

5.1.1 Calling Convention and Data Types

The CANopen Cleient API DLL exports its C API functions compliant to the '**stdcall' calling convention**. This standard is supported by nearly all compiler types and visualization tools. Basically the parameter succession on the stack and naming of the functions in the export table are defined by the calling convention.

The API functions appear in the **export table** of the DLL as:

'_<FunctionName>@<NumberOfParameterBytes>'

Examples: *CMA_InitializeCard@20*

CMA_Startup@4

CMA_InitDownload@24

The naming convention in the export table must be observed if the application directly accesses the DLL, i.e. if the supplied import library *CANopenL2.lib* is not used.

The API functions use the basic data types. Pointers of any type represent a 4-byte address. Each parameter requires 4 bytes at a 4byte aligned offset even if the actual size of a parameter is smaller (e.g. 1 byte for "unsigned char"). Actual parameter length of each function can be found in *readme.txt*.

Event Processing

5.1.2 Interrupt Events

For many applications it is useful to evaluate occurred CANopen events by an interrupt triggered routine. Otherwise, the *Event-FIFO* must be polled for new events.

When the CANopen Client API is triggered by hardware interrupts it determines the corresponding CANopen events and puts them in the *Event-FIFO*, i.e.:

- Reception of an PDO
- Reception of an EMCY
- Completion of start-up or shutdown of the Client
- Overrun of the Event-FIFO
- API internal errors
- Bus state change
- Overrun of transmit queue
- Overrun of receive queue
- SDO transfer error
- Guarding error

5.1.3 WIN32 Interrupt Programming

The CANopen Client API can trigger a WIN32 event which may be evaluated by the application to control a WIN32 process or thread. Thus, an application or thread can be created which is only processed in case of a new CANopen event.

As a prerequisite the CANopen Client API must be supplied with the handle of an application created WIN32 event by the API function *CANPC_set_interrupt_event*. Furthermore a thread should be created and started which gets into WAIT status until this event is triggered. Then, the necessary activities can be processed and the thread gets back into WAIT status.

Before any termination of the WIN32 process the created resources should be released for proper operation.

The interrupt usage is exemplarily implemented in '*Interrupt.c*' in the '*Sample**C*' directory of the installed software. This C source code provides macro functions for initialization and termination of the interrupt handling as well as an interrupt service thread.

If you setup an own application implementing the interrupt thread as demonstrated the C/C++ compiler must generate the code using the runtime library for multithread applications.

6 Function Reference

6.1 CMA_AbortSDOTransmission

Description

Calling *CMA_AbortSDOTransmission* a pending segmented SDO transfer is cancelled and the abort transfer request is sent to the Slave of Node-ID *ucNodeID*.

The command is not acknowledged to the application by the CANopen Client API.

Application Notes

CMA_AbortSDOTransmission requires an active Client in OPERATIONAL or PREOPERATIONAL state.

Function Call

short CMA_AbortSDOTransmission(

unsigned char ucChannelHandle,
unsigned char ucNodeID);

Parameter List

Parameter	Description
ucChannelHandle	CAN channel handle returned by *CMA_InitializeCard*.
ucNodeID	Node-ID of the target node. 0: CANopen Client [1-127]: Node-ID (Node-ID of CANopen Client defined by *CMA_InitNodeMan* is also possible)

Function Return Code

Return Code	Description
0:	Function completed successfully. Command posted to the Command FIFO.
-105:	*ucChannelHandle* not defined.
-117:	Command FIFO is full. Command denied.

Related Events

Event	Description
SDOError	Error in SDO transfer.

See Also

4.3 SDO Transfer
6.10 CMA_InitDownload
6.10 CMA_InitUpload
6.15 CMA_ReadEvent

6.2 CMA_AddNode

Description

CMA_AddNode introduces a CANopen Slave with Node-ID *ucNodeID* to the NMT Master and configures its Node Guarding / Heartbeat supervision parameters.

Node Guarding is parameterized by the guarding cycle time *usGuardTime*. If the Slave does not respond within this guard time a guard error event is posted to the application. The parameter *ucRetryFactor* determines the guard retries on occurrence of a guard error, i.e.

ucRetryFactor = NoOfRetries + 1.

The parameter *ucErrAction* defines whether the guarding is restarted or stopped if the guard retries are expired without a response of the Slave.

Heartbeat supervision is parameterized by the parameter *ulHeartBeatInterval*. If the Slave does not respond within this interval guard error event is posted to the application.

The function *CMA_AddNode* is a prerequisite for all subsequent NMT Master functions concerning the Slave node, i.e. node guarding and remote state change.

The added node can be removed from the NMT Masters node list by *CMA_RemoveNode.* It is automatically removed by calling *CMA_Shutdown* or by calling *CMA_Startup* again.

Application Notes

CMA_AddNode requires an active Client (start-up).

If Node Guarding or Heartbeat supervision is enabled for the added node it can be started and stopped by *CMA_RequestGuarding*. The COB-ID of the error control object is set to 1792 + Node-ID as defined in the Predefined Connection Set [CiA 301].

Function Call

```
short CMA_AddNode(
        unsigned char       ucChannelHandle,
        CMA_NodeConfig      stNodeConfig);
```

Parameter List

```
typedef struct
{
        unsigned char   ucNodeID;
        unsigned short  usGuardTime;
        unsigned char   ucRetryFactor;
        unsigned long   ulHeartBeatInterval;
        unsigned char   ucErrAction;
} CMA_NodeConfig;
```

Parameter	Description
ucChannelHandle	CAN channel handle returned by *CMA_InitializeCard*.
ucNodeID	Node-ID of the Slave node. Range: [1-127]
usGuardTime	Guarding time [ms] of the Slave node. 0: Guarding disabled. others: Guard time in ms.
ucRetryFactor	Number of guarding tries if the Slave doesn't answer. *ucRetryFactor* = NoOfRetries + 1 Range: [1-127]
ulHeartBeatInterval	Heartbeat supervision time [ms] for the Slave node. Heartbeat production time for the Client. 0: Heartbeat (supervision) disabled. others: Interval in ms.

Parameter	Description
ucErrAction	Error handling in case of detected guarding errors. Bit 0 = 0: Application is informed by a *GuardError* event in the Event-FIFO. Bit 0 = 1: Application is informed by a *GuardError* event in the Event-FIFO. Guarding is restarted if *usGuardTime* $\neq 0$. other Bits: Reserved. (initialize with 0)

Function Return Code

Return Code	Description
0:	Function completed successfully.
-103:	Function timeout.
-105:	*ucChannelHandle* not defined.
-112:	Node-ID invalid or assigned to the Client.
-113:	Client not active. Start-up required.

Related Events

Event	Description
GuardError	Guarding error occured. (see section 4.6.2)

Example

NMTServices.c

See Also

4.6.1 Node Manager Configurations
6.19 CMA_RemoveNode

6.3 CMA_ChangeState

Description

CMA_ChangeState is used to change the state of a Slave or the Client, e.g. state change to OPERATIONAL to enable PDO transfer.

The service initiates a state transition of a certain node with Node-ID *ucNodeID* or of all nodes simultaneously. The new state is determined by *ucState*.

Application Notes

Calling *CMA_ChangeState* requires an active Client (Start-up).

Before using *CMA_ChangeState* the selected Slave node must have been registered by *CMA_AddNode*. The Node-ID of the Client is 127 or defined by *CMA_InitNodeMan*. In case of a Client state transition the sending of the NMT service object (COB-ID=0) is suppressed.

A Client state transition to OPERATIONAL is necessary for the PDO transfer. It is realized by *CMA_ChangeState* referring to the Client Node-ID defined by *CMA_ConfigNodeMan*.

Function Call

```
short CMA_ChangeState(
                unsigned char   ucChannelHandle     ,
                unsigned char   ucNodeID,
                unsigned char   ucState);
```

Parameter List

Parameter	Description
ucChannelHandle	CAN channel handle returned by *CMA_InitializeCard*.
ucNodeID	Node-ID of the Slave node. 0: CANopen Client [1-127]: Single Node-ID 128: All nodes
ucState	Requested state (*state transition*) 4: STOPPED (for compatibility with CiA 301 V3.0 also called “PREPARED”) (*Stop Remote Node*) 5: OPERATIONAL (*Start Remote Node*) 6: RESET APPLICATION (*Reset Node*) 7: RESET COMMUNICATION (*Reset Communication*) 127: PRE-OPERATIONAL (*Enter Pre-operational*)

Function Return Code

Return Code	Description
0:	Function completed successfully.
-103:	Function timeout.
-105:	*ucChannelHandle* not defined.
-109:	Node-ID *ucNodeID* not valid (*CMA_AddNode*).
-113:	Client not active. Start-up required.
-114:	Requested state *ucState* is not defined.
-115:	Transmit queue is full.

Related Events

None.

Example

NMTServices.c

See Also

4.6.1 Node Manager Configuration
4.6.3 Module State Control
6.2 CMA_AddNode
6.19 CMA_RemoveNode

6.4 CMA_CloseCard

Description

CMA_CloseCard releases the hardware driver resources and the handle *ucCANHandle* allocated by *CMA_InitializeCard*.

Application Notes

CMA_CloseCard is a mandatory function call before any exit of the application. If the function is omitted the resources may stay locked and calling *CMA_InitializeCard* subsequently may fail.

Before leaving the CANopen Client application by *CMA_CloseCard* it should be shutdown by *CMA_Shutdown* to prevent the network from uncontrolled actions and uncertain behavior.

Function Call

```
short CMA_CloseCard(
                unsigned char       *ucpCANHandle);
```

Parameter List

Parameter	Description
ucpCANHandle	Handle for CAN access returned by *CMA_InitializeCard*.

Function Return Code

Return Code	Description
0:	Function completed successfully.
-105:	CAN channel handle not defined.

Related Events

None.

Example

```
static unsigned char  ucCANHandle;


// Allocate CAN channel named “CANpro USB_1”
CMA_InitializeCard (0xFFFFFFFF, 1, &ucCANHandle,
0,”CANpro USB_1”);

// start-up Master  with new settings
ret = CMA_Startup (ucCANHandle);

        ...

// shutdown Master
ret = CMA_Shutdown (ucCANHandle);

// Release driver resources
ret = CMA_CloseCard (ucCANHandle);
```

See Also

4.2 Initialization
6.8 CMA_InitializeCard

6.5 CMA_ConfigCANChannel

Description

The function *CMA_ConfigCANChannel* defines the

Baudrate and

Acceptance filter

settings of the CAN channel referenced by *ucChannelHandle*.

The defined parameter values are buffered in the CANopen Client API. They are applied to the CAN controller during the next start-up of the CANopen Client (*CMA_Startup*).

The bit timing *ucBaudrate* can be set to certain predefined values according to the CANopen recommendation (see parameter table).

CMA_ConfigCANChannel can be used optionally. If it is omitted or *CMA_Shutdown* was called afterwards, the parameters are set to following default values:

Baudrate = 125 kbit/s

No acceptance filter (*ucAccMask* = *ucAccCode* = 0), all identifiers are received.

Application Notes

CMA_ConfigCANChannel can be called at any time after initialization of the CAN channel (*CMA_InitializeCard*) but the changed values are taken into effect by the next Client start-up (*CMA_Start_up*).

If Baudrate *ucBaudrate* is set to 0xFF it is possible to define the bit time configuration individually by the parameters *ucPresc, ucSjw; ucTSeg1* and *ucTSeg2*.

It is recommended not to use acceptance filtering. If *ucAccMask* is set to values other than 0, identifiers not matching with *ucAccCode* in their eight most significant bits are not received. This may affect the compliance with the CANopen protocol specification. Therefore acceptance filtering should only be used by experts.

Function Call

```
short CMA_ConfigCANChannel(
        unsigned char       ucChannelHandle,
        CMA_CAN_CONFIG      stCANConfig);
```

Parameter List

```
typedef struct
 {
        unsigned char       ucBaudrate;
        unsigned char       ucPresc;
        unsigned char       ucSjw;
        unsigned char       ucTSeg1;
        unsigned char       ucTSeg2;
        unsigned char       ucAccCode;
        unsigned char       ucAccMask;
 }  CMA_CAN_CONFIG;
```

Parameter	Description
ucChannelHandle	CAN channel handle returned by *CMA_InitializeCard*.
ucBaudrate	Predefined baudrate. 0: 1 Mbit/s 1: 800 Kbit/s 2: 500 Kbit/s 3: 250 Kbit/s 4: 125 Kbit/s 5: 50 Kbit/s 6: 20 Kbit/s 7: 10 Kbit/s 0xFF: No predefined baudrate. ucPresc, ucSjw, ucTSeg1 and TSeg2 are valid
	others: Not valid

Parameter	Description
ucPresc	Prescaler ; only valid with ucBaudrate=0xFF
ucSjw	Synchronization jump width, only valid with ucBaudrate=0xFF.
ucTSeg1	Time segment 1; only valid with ucBaudrate=0xFF.
ucTSeg2	Time segment 1; only valid with ucBaudrate=0xFF.
ucAccCode	Acceptance code only valid with ucBaudrate=0xFF.
ucAccMask	Acceptance mask only valid with ucBaudrate=0xFF.

Function Return Code

Return Code	Description
0:	Function completed successfully.
-103:	Function timeout.
-105:	*ucChannelHandle* not defined.
-106:	Invalid baudrate parameter

Related Events

None.

Example

```
// Setting baudrate to 1Mbit/s,

// ucCANHandle returned by CMA_initializeCard

CMA_CAN_CONFIG    CANConfig;

CANConfig.ucBaudRate = 0;
ret = CMA_ConfigCANChannel (ucCANHandle, CANConfig);

// Reset and start Client (CAN controller with new settings)
ret = CMA_Startup (ucCANHandle);
```

See Also

4.2.3 Start-up and Shutdown of the CANopen Client
6.25 CMA_StartUp

6.6 CMA_ConfigSyncMan

Description

CMA_ConfigSyncMan configures the SYNC Manager of the CANopen Client.

The identifier of the SYNC object *ulSyncID* and its cycle period in µs *ulSyncCycle* are defined in the parameter structure *stSyncManConfig.*

If the Client is configured as SYNC producer by *ucSendMode=1* the SYNC object is sent periodically until the SYNC manager is reconfigured with *ucSendMode=0*.

If the Client application is intended to transfer synchronous PDOs the consumer mode has to be enabled setting *ucConsumerMode=1*.

The Client can act as a SYNC producer and consumer at the same time.

Application Notes

During the start-up of the Client the SYNC manager is automatically configured to the default settings. Thus, it neither produces nor consumes any SYNC object. For definition and start of the SYNC handling *CMA_ConfigSyncMan* can be called at any time after the successful Client start-up.

The default COB-ID of the SYNC object is defined to 0x80 by the Predefined Connection Set of CANopen [CiA 301] but can be changed at customer's choice.

Function Call

```
short CMA_ConfigSyncMan(
                unsigned char           ucChannelHandle
                CMA_SyncManConfig       stSyncManConfig);
```

Parameter List

```
typedef struct
{
        unsigned char   ucSendMode;
        unsigned char   ucConsumerMode;
        unsigned long   ulSyncID;
        unsigned long   ulSyncCycle;
} CMA_SyncManConfig;
```

Parameter	Description
ucChannelHandle	CAN channel handle returned by *CMA_InitializeCard*.
ucSendMode	SYNC producer mode of the Client 1: Client sends the SYNC object. 0: Client does not send the SYNC object. (default)
ucConsumerMode	SYNC consumer mode of the Client. 1: Client consumes the SYNC object (necessary for handling of synchronous PDOs). 0: Client doesn't consume the SYNC object
ulSyncID	COB-ID of the SYNC Default: 0x80
ulSyncCycle	Period [µs] of the cyclic transmission of the SYNC object. (resolution 1µs) Default: 0 Maximum: 32767000

Function Return Code

Return Code	Description
0:	Function completed successfully.
-103:	Function timeout.
-105:	*ucChannelHandle* not defined.
-110:	COB-ID invalid or already assigned to another object.
-111:	Parameter error: *ulSyncCycle* > 32767000

Related Events

None.

Example

PDOTransfer.c

See Also

4.4.1	PDO Buffer Configuration
4.4.2	PDO Services
6.15	CMA_InstallPDO_E
6.26	CMA_WritePDO
6.18	CMA_ReadPDO

6.7 CMA_GetVersion

Description

CMA_GetVersion provides version information of the applied hardware and software to the application:

- Name of the applied hardware
- Serial number of the hardware
- Hardware revision
- Software version

Application Notes

CMA_GetVersion can be called at any time after hardware initialization by *CMA_InitializeCard*.

The hardware revision number is split into a main and a sub number. The software version of the CANopen Client API is split into a major, a minor and a build number.

Function Call

```
short CMA_GetVersion(
                unsigned char       ucChannelHandle ,
                CMA_VersionStruct *stpVersion);

typedef struct
{
  unsigned long  ulHWSerialNumber;
  unsigned char  ucHWMainRevision;
  unsigned char  ucHWSubRevision;
  unsigned char  ucHWIdentString[18];
  unsigned char  ucSWMajorVersion;
  unsigned char  ucSWMinorVersion;
  unsigned char  ucSWBuild;
 } CMA_VersionStruct;
```

Parameter List

Parameter	Description
ucChannelHandle	CAN channel handle returned by *CMA_InitializeCard*.
ulHWSerialNumber	Serial number of the CAN interface. Example: 094305678
ucHWMainRevision	Hardware revision number - Main number Example: 1.**
ucHWSubRevision	Hardware revision number - Sub number Example: *.01
ucHWIdentString[18]	Hardware name. Example: CAN-AC1-PCI, CANpro USB
ucSWMajorVersion	Software version number - Main number Example: 1.**
ucSWMinorVersion	Software version number - Sub number Example: *.10
ucSWBuild	Software version number - Build number Example: 4

Function Return Code

Return Code	Description
0:	Function completed successfully.
-105:	*ucChannelHandle* not defined.

6.8 CMA_InitializeCard

Description

CMA_InitializeCard allocates the specified CAN channel. It provides a handle to the CAN channel.

The CAN channel is selected by its channel name. This name can be defined using the Softing CAN Interface Manager (Click *Start – All Programs – Softing CAN – Runtime System Configuration – Softing CAN Interface Manager*).

Application Notes

CMA_InitializeCard is a mandatory initialization call before any usage of other API functions.

If the function fails the returned error code provides useful information about the error cause.

The returned CAN channel handle is only valid if the function returns without error.

Function Call

```
short CMA_IntializeCard(
                unsigned long   ulDeviceIndex,
                unsigned char   ucCANFunctionality,
                unsigned char   *ucpCANHandle,
                unsigned char   ucLicense,
                unsigned char   *pChannelName
);
```

Parameter List

Parameter	Description
ulDeviceIndex	Device index. FFFFFFFFH: Search by ChannelName others: Not valid.
ucCANFunctionality	CAN channel operation mode 1: CANopen Client mode others: Not valid
ucpCANHandle	Returned handle for CAN access (starting with 0 for the first channel)
ucLicense	License type 0: dummy
pChannelName	Name of CAN channel to be used

Function Return Code

Return Code	Description
0:	Function completed successfully, CAN access provided.
-100:	For compatibility only.
-101:	For compatibility only.
-102:	EEPROM access error
-103:	Function timeout.
-104:	Parameter error in *ucCANFunctionality* or in *ucCANFunctionality*.
-500:	For compatibility only.

-501:	Not enough memory to register the interface.
-502:	Driver not found.
-503:	False driver version.
-504:	False driver DLL version.
-505:	Internal error: Functionality not supported.
-506:	Internal error: Illegal driver call.
-507:	Internal error: Driver call was cancelled.
-508:	Internal error: Driver call still pending.
-509:	Internal error: Driver call timeout.
-510:	Internal error: General driver call error.
-511:	Internal driver error.
-512:	No CAN device found.
-513:	Unknown device.
-514:	Device already registered by the driver.
-515:	Device already opened.
-516:	Resources already in use (DPRAM, IRQ, I/O).
-517:	Resource conflict
-518:	Resource access error.
-519:	Internal error: Invalid memory access.
-520:	Internal error: To many I/O ports requested..
-521:	Internal error: Invalid resource access
-522:	Driver DLL not found.
-523:	Interface could not be registered.
-524:	Invalid interrupt number or interrupt not available.
-550:	Internal error: Handle invalid.
>0:	For compatibility only.

Related Events

None.

Example

```
static unsigned char  ucCANHandle;

static unsigned char ChannelName[] = "CANpro USB_1";


CMA_InitializeCard (0xFFFFFFFF, 1, &ucCANHandle, 0,
ChannelName);
```

See Also

4.2 Initialization
6.4 CMA_CloseCard

6.9 CMA_InitBlockDownload

Description

CMA_InitBlockDownload starts the block download of an SDO to the Object Dictionary of the CANopen node addressed by Node-ID *ucNodeID*. The SDO is referenced by its index and sub-index as defined in the Predefined Connection Set [CiA 301].

The CANopen Client API autonomously performs the block SDO download protocol for the supplied data.

Successful block SDO downloads are confirmed by the event *DownloadCompleted* in the *Event-FIFO*.

Timeout control

The SDO transfer is observed by timers configured by *CMA_InitSDOMan*. A timeout results in an *SDOError* event in the *Event-FIFO*:

If necessary the SDO download can be aborted by the application calling *CMA_AbortSDOTransfer*:

Possible SDO transfer errors are signalled by the *SDOError* event including the error code (see Appendix A).

Application Notes

Calling *CMA_InitBlockDownload* requires an active Client (*CMA_Startup*).

The block SDO transfer uses the default SDOs defined by the Predefined Connection Set [CiA 301] only.

Function Call

short CMA_InitBlockDownload(

unsigned char	*ucChannelHandle ,*
unsigned char	*ucNodeID,*
unsigned short	*usIndex,*
unsigned char	*ucSubIndex,*
unsigned long	*ulTotalDataSize,*
unsigned char	**ucpData);*

Parameter List

Parameter	Description
ucChannelHandle	CAN channel handle returned by *CMA_InitializeCard*.
ucNodeID	Node-ID of the target device. 0: CANopen Client 1-127: Target Node-ID (Client ID possible)
usIndex	SDO index in the object dictionary.
ucSubIndex	SDO subindex in the object dictionary.
ulTotalDataSize	Number of bytes.
ucpData	Pointer to the data buffer.

Function Return Code

Return Code	Description
0:	Function completed successfully. Command posted to the Command FIFO.
-105:	*ucChannelHandle* not defined.
-117:	Command FIFO is full.

Related Events

Event	Description
DownloadCompleted	SDO download successfully completed.
SDOError	Error during SDO transfer (including error code).

Example

SDODownload.c

See Also

4.3 SDO Transfer
4.3.2 SDO Download
6.14 CMA_InitSDOMan
6.1 CMA_AbortSDOTransfer
6.15 CMA_ReadEvent

6.10 CMA_InitBlockUpload

Description

CMA_InitBlockUpload starts the block upload of an SDO from the Object Dictionary of the CANopen node addressed by Node-ID *ucNodeID*. The SDO is referenced by its index and sub-index as defined in the Predefined Connection Set [CiA 301].

The data segments are uploaded using the block SDO transfer protocol. The maximum number of data segments within a confirmed block is determined by the parameter *ucBlockSize*. The CANopen Client API autonomously takes care about buffering the uploaded data. The successful upload is confirmed by the event *UploadCompleted* in the *Event-FIFO.* The event contains length information about the uploaded data. The uploaded data then need to be read by the application program calling *CMA_ReadData*.

Timeout control

The SDO transfer is observed by timers configured by *CMA_InitSDOMan*. A timeout results in an *SDOError* event in the *Event-FIFO*:

If necessary the SDO upload can be aborted by the application calling *CMA_AbortSDOTransfer*:

Possible SDO transfer errors are signalled by the *SDOError* event including the error code (see Appendix A).

Application Notes

Calling *CMA_InitBlockUpload* requires an active Client (*CMA_Startup*).

The SDO transfer uses the default SDOs defined by the Predefined Connection Set [CiA 301] only.

Only one block SDO upload per Node-ID may be active at a time. SDO uploads for different Node-IDs can take place in parallel.

The parameter ucBlockSize is just supplied as information to the protocol engine inside the CANopen Client API for proper handling of the SDO block upload. It does not limit the total data lengths of the upload.

The CANopen Client API dynamically allocates memory for buffering uploaded data of the SDO block transfer.

Function Call

short CMA_InitBlockUpload(

unsigned char	*ucChannelHandle ,*
unsigned char	*ucNodeID,*
unsigned short	*usIndex,*
unsigned char	*ucSubIndex,*
unsigned char	*ucBlockSize);*

Parameter List

Parameter	Description
ucChannelHandle	CAN channel handle returned by *CMA_InitializeCard*.
ucNodeID	Node-ID of the target device. 0: CANopen Client 1-127: Target Node-ID (Client ID possible)
usIndex	SDO index in the object dictionary.
ucSubIndex	SDO subindex in the object dictionary.
ucBlockSize	Maximum number of data segments within a confirmed block.

Function Return Code

Return Code	Description
0:	Function completed successfully. Command posted to the Command FIFO.
-105:	*ucChannelHandle* not defined.
-117:	Command FIFO is full.

Related Events

Event	Description
UploadCompleted	SDO upload successfully completed.
SDOError	Error during SDO transfer (including error code).

Example

SDOUpload.c

See Also

4.3 SDO Transfer
4.3.3 SDO Upload
6.14 CMA_InitSDOMan
6.1 CMA_AbortSDOTransfer
6.15 CMA_ReadEvent
6.16 CMA_ReadData

6.11 CMA_InitDownload

Description

CMA_InitDownload starts the download of an SDO to the Object Dictionary of the CANopen node addressed by Node-ID *ucNodeID*. The SDO is referenced by its index and sub-index as defined in the Predefined Connection Set [CiA 301].

The CANopen Client API evaluates the data length *ulTotalDataSize* and decides autonomously if the data fit into an expedited SDO download or if a segmented SDO download protocol is required.

Successful SDO downloads, regardless of their type (expedited or segmented) are confirmed by the event *DownloadCompleted* in the *Event-FIFO*.

Timeout control

The SDO transfer is observed by timers configured by *CMA_InitSDOMan*. A timeout results in an *SDOError* event in the *Event-FIFO*:

If necessary the SDO download can be aborted by the application calling *CMA_AbortSDOTransfer*:

Possible SDO transfer errors are signalled by the *SDOError* event including the error code (see Appendix A).

Application Notes

Calling *CMA_InitDownload* requires an active Client (*CMA_Startup*).

The SDO transfer uses the default SDOs defined by the Predefined Connection Set [CiA 301] only.

Function Call

short CMA_InitDownload(

unsigned char	*ucChannelHandle ,*
unsigned char	*ucNodeID,*
unsigned short	*usIndex,*
unsigned char	*ucSubIndex,*
unsigned long	*ulTotalDataSize,*
unsigned char	**ucpData);*

Parameter List

Parameter	Description
ucChannelHandle	CAN channel handle returned by *CMA_InitializeCard*.
ucNodeID	Node-ID of the target device. 0: CANopen Client 1-127: Target Node-ID (Client ID possible)
usIndex	SDO index in the object dictionary.
ucSubIndex	SDO subindex in the object dictionary.
ulTotalDataSize	Number of bytes.
ucpData	Pointer to the data buffer.

Function Return Code

Return Code	Description
0:	Function completed successfully. Command posted to the Command FIFO.
-105:	*ucChannelHandle* not defined.
-117:	Command FIFO is full.

Related Events

Event	Description
DownloadCompleted	SDO download successfully completed.
SDOError	Error during SDO transfer (including error code).

Example

SDODownload.c

See Also

4.3 SDO Transfer
4.3.2 SDO Download
6.14 CMA_InitSDOMan
6.1 CMA_AbortSDOTransfer
6.15 CMA_ReadEvent

6.12 CMA_InitUpload

Description

CMA_InitUpload starts the upload of an SDO from the Object Dictionary of the CANopen node addressed by Node-ID *ucNodeID*. The SDO is referenced by its index and sub-index as defined in the Predefined Connection Set [CiA 301].

Expedited SDO transfer (≤4bytes)

If the total data size does not exceed 4 bytes the successful upload is confirmed by the event *UploadCompleted* in the *Event-FIFO*. The requested SDO data are included in this event.

Segmented SDO transfer (>4bytes)

If the total data size is greater than 4 bytes the data segments are transferred using the segmented SDO transfer protocol. The CANopen Client API autonomously takes care about buffering the uploaded data. The successful upload is confirmed by the event *UploadCompleted* in the *Event-FIFO.* The event contains length information about the uploaded data. The uploaded data then need to be read by the application program calling *CMA_ReadData*.

Timeout control

The SDO transfer is observed by timers configured by *CMA_InitSDOMan*. A timeout results in an *SDOError* event in the *Event-FIFO*:

If necessary the SDO upload can be aborted by the application calling *CMA_AbortSDOTransfer*:

Possible SDO transfer errors are signalled by the *SDOError* event including the error code (see Appendix A).

Application Notes

Calling *CMA_InitUpload* requires an active Client (*CMA_Startup*).

The SDO transfer uses the default SDOs defined by the Predefined Connection Set [CiA 301] only.

Only one segmented SDO upload per Node-ID may be active at a time. SDO uploads for different Node-IDs can take place in parallel.

The CANopen Client API dynamically allocates memory for buffering uploaded data of the SDO transfer.

Function Call

short CMA_InitUpload(

unsigned char	*ucChannelHandle ,*
unsigned char	*ucNodeID,*
unsigned short	*usIndex,*
unsigned char	*ucSubIndex);*

Parameter List

Parameter	Description
ucChannelHandle	CAN channel handle returned by *CMA_InitializeCard*.
ucNodeID	Node-ID of the target device. 0: CANopen Client 1-127: Target Node-ID (Client ID possible)
usIndex	SDO index in the object dictionary.
ucSubIndex	SDO subindex in the object dictionary.

Function Return Code

Return Code	Description
0:	Function completed successfully. Command posted to the Command FIFO.
-105:	*ucChannelHandle* not defined.
-117:	Command FIFO is full.

Related Events

Event	Description
UploadCompleted	SDO upload successfully completed.
SDOError	Error during SDO transfer (including error code).

Example

SDOUpload.c

See Also

4.3 SDO Transfer
4.3.3 SDO Upload
6.14 CMA_InitSDOMan
6.1 CMA_AbortSDOTransfer
6.15 CMA_ReadEvent
6.16 CMA_ReadData

6.13 CMA_InitNodeMan

Description

CMA_InitNodeMan initializes the node manager by means of determining the Node-ID of the CANopen Client. The chosen Node-ID is buffered in the CANopen Client API and assigned to the Client by the next start-up (*CMA_Startup*).

Application Notes

CMA_InitNodeMan can be called at any time after initialization of the CAN channel (*CMA_InitializeCard*) but the defined values are set into effect after the next Client start-up by *CMA_Startup*.

The function is applied optionally. If it is omitted the Node-ID of the Client is automatically set to 127. Apart from this Node-ID the Client always can be addressed by the API using the reserved Node-ID 0.

Function Call

```
short CMA_InitNodeMan(
                unsigned char   uChannelHandle
                unsigned char   ucNodeID);
```

Parameter List

Parameter	Description
ucChannelHandle	CAN channel handle returned by *CMA_InitializeCard*.
ucNodeID	Node-ID of the CANopen Client Range: [1..127] Default: 127

Function Return Code

Return Code	Description
0:	Function completed successfully.
-103:	Function timeout.
-105:	*ucChannelHandle* not defined.
-109:	Invalid Node-ID.

Related Events

None.

Example

```
// Configuration of the Node Manager with Node-ID=10
// ucCANHandle returned by previous CMA_InitializeCard

ret = CMA_InitNodeMan (ucCANHandle, 10);

// start-up Client  with new settings
ret = CMA_Startup (ucCANHandle);
```

See Also

4.2 Initialization
6.8 CMA_InitializeCard
6.25 CMA_StartUp

6.14 CMA_InitSDOMan

Description

This function initializes the timeout parameters of the SDO Manager. The parameter values are buffered in the CANopen Client API and are assigned to the SDO manager during the start-up of the CANopen Client (*CMA_Startup*).

Parameter *ucSDOTimeout* is the time in ms the CANopen Client waits for a Slave response to an SDO request. Parameter *usApplTimeoutFactor* is only a dummy for compatibility with earlier versions.

Application Notes

CMA_InitSDOMan can be called at any time after initialization of the CAN channel (*CMA_InitializeCard*) but the defined values are set into effect after the next Client start-up by *CMA_Startup*.

The function can be applied optionally. If it is omitted the SDO manager is initialized with the default settings during the CANopen Client start-up.

Function Call

```
short CMA_InitSDOMan(
    unsigned char   ucCANChannel,
    unsigned short  usSDOTimeout,
    unsigned short  usApplTimeoutFactor);
```

Parameter List

Parameter	Description
ucChannelHandle	CAN channel handle returned by *CMA_InitializeCard*.
usSDOTimeout	Timeout [ms] of the Slave response to an SDO request of the Client. Range: [1...65535*)] default: 2000
usApplTimeoutFactor	Prolongation factor of *usSDOTimeout* Only for compatibility reasons with earlier versions of this API. Should be set to 1 Range: [1...65535*)] default: 1

*) *Restriction: usSDOTimeout * usApplTimeoutFactor ≤ 65535*

Function Return Code

Return Code	Description
0:	Function completed successfully.
-103:	Function timeout.
-105:	*ucChannelHandle* not defined.
-107:	Maximum of 65535 ms exceeded.
-108:	Timeout parameter out of range.

Related Events

None.

Example

```
// Configuration of SDO manager with
// timeout = 1 s and a prolongation factor of  1

usSDOTimeout = 1000;
usApplTimeoutFactor=1);

ret = CMA_InitSDOMan (ucCANHandle, usSDOTimeout,
usApplTimeoutFactor);

// start-up Client with new settings
ret = CMA_Startup (ucCANHandle);
```

See Also

4.2.3 Start-up and Shutdown of the CANopen Client
4.3 SDO Transfer
6.8 CMA_InitializeCard
6.25 CMA_StartUp
6.9 CMA_InitBlockDownload
6.10 CMA_InitBlockUpload
6.11 CMA_InitDownload
6.12 CMA_InitUpload

6.15 CMA_InstallPDO_E

Description

CMA_InstallPDO_E installs a PDO object in the CANopen Client API.

Overall, 512 transmit PDO objects (TPDOs) and 512 receive PDO objects (RPDOs) are configurable

The function returns the handle to the PDO object *uspPDOHandle* which is required for the PDO access functions.

In the parameter structure *stPDOConfig_E* the COB-ID *ulCOBID* and the transmission type *ucTransType* of the PDO are defined. Initial values of the PDO data are configured by *ucaDefVal* and *ucNrOfByte*.

If an installed RPDO is received the CANopen Client API informs the application by the event *PDOReceived* in the Event-FIFO and triggers an interrupt event to the application. The notification of the application about the reception of an RPDO can be switched on/off individually for each RPDO by *ucEventNotify*.

Installed PDOs are removed either individually by *CMA_RemovePDO* or globally by *CMA_Startup* and *CMA_Shutdown*.

Application Notes

Installing a PDO requires an active Client (*CMA_Startup*).

If an asynchronous TPDO is installed and the Client is in OPERATIONAL state *CMA_InstallPDO_E* transmits the PDO with its initial data. Similarly, if the Client is switched from PRE-OPERATIONAL to OPERATIONAL state all previously installed TPDOs are sent once. The initial transmission of TPDO data may be suppressed by the parameter *ucNoInitTrans*.

The parameters *ucAccType* and *usEventTimer* are dummies intended to be used in future API versions.

The PDO transmission type *ucTransType* conforms to the pre-definitions of the PDO type number in the CANopen specification (see Table 6-1).

Table 6-1: PDO transmission type

Transmission type	PDO transmission				
	cyclic	acyclic	sync.	async.	RTR only
0		X	X		
1-240	X		X		
241-251	reserved				
252			X		X
253				X	X
254				X	
255				X	

Function Call

short CMA_InstallPDO_E(

unsigned char	*ucChannelHandle,*
CMA_PDOConfig_E	*stPDOConfig_E,*
unsigned short	**uspPDOHandle);*

Parameter List

```
typedef struct
{
        unsigned char   ucPDOType;
        unsigned long   ulCOBID;
        unsigned char   ucTransType;
        unsigned char   ucAccType;
        unsigned char   ucEventNotif;
        unsigned char   ucaDefVal[8];
        unsigned short  usEventTimer;
       unsigned char   ucNrOfByte;
       unsigned char   ucNoInitTrans;

} CMA_PDOConfig_E;
```

Parameter	Description
ucChannelHandle	CAN channel handle returned by *CMA_InitializeCard*.
uspPDOHandle	Returned handle to the PDO object. The handle is used by other PDO functions.
ucPDOType	PDO type. 0: RX-PDO others: TX-PDO
ulCOBID	COB-ID used by the PDO.
ucTransType	Transmission/reception type of the PDO defined in [CiA 301] (see Table 6-1).
ucAccType	Dummy for future use. 0: default others: Invalid.

Parameter	Description
ucEventNotif	Enable event *PDOReceived* to the application if the RPDO is received. 0: No event in Event-FIFO. others: Event *PDOReceived* enabled.
ucaDefVal[8]	Array with initial values of the PDO data.
usEventTimer	Dummy for future use. 0: default others: Invalid.
ucNrOfByte	Number of data bytes of the PDO.
ucNoInitTrans	Inhibit flag for initial PDO transmission.
	0: Initial transmission is active. others: No initial transmission

Function Return Code

Return Code	Description
0:	Function completed successfully.
-103:	Function timeout.
-105:	*ucChannelHandle* not defined.
-110:	COB-ID invalid or already assigned to another object.
-113:	Client not active. Start-up required.
-125	Parameter *ucAccType* invalid (must be 0).

Related Events

Event	Description
PDOReceived	RPDO received. PDO handle, data length and values are provided by the event.

Example

PDOTransfer.c

See Also

4.4 PDO Transfer
6.20 CMA_RemovePDO

6.16 CMA_ReadData

Description

CMA_ReadData reads uploaded SDO data after the completion of an SDO upload regardless of its type (segmented, block or expedited).

The CANopen Client API provides the uploaded data by entering them into an application supplied buffer.

Application Notes

CMA_ReadData can be called at any time after a successful SDO upload, i.e. the occurrence of event *UploadCompleted* in the *Event-FIFO*.

For each Node-ID there is a single internal buffer that is used by the CANopen Client API for buffering SDO uploads. For a given Node-ID only the latest SDO upload data are available. All data of earlier uploads are overwritten by succeeding uploads from that node.

Uploaded data can only be read if *SubIndex* and *Index* match the information given in the event data of the event *UploadCompleted*.

The application supplied data buffer must be large enough to carry the number of bytes indicated by the parameter *ulDataLenSegmented* in the event *UploadCompleted*.

Function Call

```
short CMA_ReadData(
                    unsigned char     ucChannelHandle,
                    unsigned char     ucNodeID,
                    unsigned char     ucSubIndex,
                    unsigned short    usIndex,
                    unsigned long     ulBufferSize,
                    unsigned char     *pData);
```

Parameter List

Parameter	Description
ucChannelHandle	CAN channel handle returned by *CMA_InitializeCard*.
ucNodeID	Node-ID of the device from which uploaded data are to be read. 0: CANopen Client 1-127: Target Node-ID (Client ID possible)
ucSubIndex	SDO subindex in the object dictionary.
usIndex	SDO index in the object dictionary.
ulBufferSize	Size of the buffer pointed to by *pData*.
pData	Pointer to the data buffer.

Function Return Code

Return Code	Description
0:	Function completed successfully.
-105:	*ucChannelHandle* not defined.
-119:	*ulBufferSize* is too small for current data.
-120:	*usIndex* does not match the index of the buffered data.
-121:	*ucSubIndex* does not match the subindex of the buffered data.
-135:	An active upload for the specified Node-ID is in progress.

Related Events

None.

Example

SDOUpload.c

See Also

4.7 API Events and Event-FIFO
4.3 SDO Transfer
4.3.3 SDO Upload

6.10 CMA_InitBlockUpload
6.12 CMA_InitUpload
6.14 CMA_InitSDOMan
6.15 CMA_ReadEvent

6.17 CMA_ReadEvent

Description

CMA_ReadEvent reads the next CANopen Client event from the *Event-FIFO*.

Type and data of the event are provided in the returned parameter structure **stpEvent* (see section 4.7.2).

Application Notes

CMA_ReadEvent can be called at any time after the initialization of the CAN channel by *CMA_InitializeCard*.

If an interrupt event is installed (see section 0) this interrupt is triggered on every occurrence of an event. Thus, the application can read the *Event-FIFO* within a thread. Otherwise, the *Event-FIFO* must be polled for new events.

Function Call

```
short CMA_ReadEvent(
                unsigned char       ucChannelHandle,
                CMA_EventStruct     *stpEvent);
```

Parameter List

```
typedef struct
{
        unsigned char   ucEventType;
        unsigned long   ulTimeStamp;
        unsigned char   ucaEventData[16];
        unsigned long   ulDataLenSegmented;

} CMA_EventStruct;
```

Parameter	Description
ucChannelHandle	CAN channel handle returned by *CMA_InitializeCard*.
ucEventType	Event type (see section 4.7.1) 0: *NoEvent* 3: *DownloadCompleted* 5: *UploadCompleted* 11: *EMCYReceived* 15: *ErrorEvent* 30: *NextDownloadSegment* (for compatibility only) 31: *NextUploadSegment* (for compatibility only) 32: *PDOReceived* 33: *ShutdownCompleted* 34: *StartupCompleted*
ulTimeStamp	Not implemented yet (invalid).
ucaEventData	Event related data of interest. (see section 4.7.2)
ulDataLenSegmented	Length information in case of the event *UploadCompleted*

Function Return Code

Return Code	Description
0:	Function completed successfully.
-105:	*ucChannelHandle* not defined.
-131:	Unknown event in *Event-FIFO*.

Related Events

None.

Example

SDODownload.c
SDOUpload.c
PDOTransfer.c

See Also

4.7 API Events and Event-FIFO
6.23 CMA_SetIntEvent

6.18 CMA_ReadPDO

Description

CMA_ReadPDO reads the current data of the RPDO or TPDO referenced by *ucPDOHandle*.

The PDO data are copied to the first 8 data bytes of the data array referenced by *ucpData.*

Application Notes

Reading the PDOs requires the CANopen Client to be active (*CMA_Startup*) and a valid PDO handle *ucPDOHandle* previously provided by *CMA_InstallPDO_E*.

If the PDO was not transferred yet the data are equal to the initial data set by *CMA_InstallPDO_E*.

Function Call

```
short CMA_ReadPDO(
                unsigned char     ucChannelHandle,
                unsigned short    usPDOHandle,
                unsigned char     *ucpData);
```

Parameter List

Parameter	Description
ucChannelHandle	CAN channel handle returned by *CMA_InitializeCard*.
usPDOHandle	PDO handle provided by *CMA_InstallPDO_E*.
ucpData	Pointer to the PDO data array.

Function Return Code

Return Code	Description
0:	Function completed successfully.
-103:	Function timeout.
-105:	*ucChannelHandle* not defined.
-126:	PDO handle not valid.
-129:	Error accessing PDO

Related Events

None.

Example

PDOTransfer.c

See Also

4.4 PDO Transfer
4.4.2 PDO Services
6.15 CMA_InstallPDO_E
6.20 CMA_RemovePDO
6.27 CMA_WritePDO
6.28 CMA_WritePDOBit
6.22 CMA_SendRemotePDO

6.19 CMA_RemoveNode

Description

CMA_RemoveNode switches the Slave *ucNodeID* to the state *ucState* and removes it from the node list of the node manager.

Application Notes

CMA_RemoveNode requires an active CANopen Client and a valid Node-ID introduced by *CMA_AddNode*.

Function Call

short CMA_RemoveNode(

unsigned char ucChannelHandle ,
unsigned char ucNodeID,
unsigned char ucState);

Parameter List

Parameter	Description
ucChannelHandle	CAN channel handle returned by *CMA_InitializeCard*.
ucNodeID	Node-ID of the Slave node. Range: [1-127] except Node-ID of the Client.
ucState	Requested state of the Slave node after removing it. 4: STOPPED (for compatibility with CiA 301 V3.0 also called “PREPARED”) (*Stop Remote Node*) 5: OPERATIONAL (*Start Remote Node*) 6: RESET APPLICATION (*Reset Node*)

	7: RESET COMMUNICATION (*Reset Communication*) 127: PRE-OPERATIONAL (*Enter Pre-operational*)

Function Return Code

Return Code	Description
0:	Function completed successfully.
-103:	Function timeout.
-105:	*ucChannelHandle* not defined.
-109:	*ucNodeID* not valid or not defined by *CMA_AddNode*.
-113:	Client not active. Start-up required.
-114:	Requested state *ucState* is not defined.
-115:	Transmit queue is full. Transmission of the state change request unsuccessful.

Related Events

None.

Example

NMTServices.c

See Also

4.6 NMT Services
4.6.1 Node Manager Configuration
6.2 CMA_AddNode

6.20 CMA_RemovePDO

Description

CMA_RemovePDO removes an installed TPDO or RPDO.

Application Notes

Removing the PDO requires the CANopen Client to be active (*CMA_Startup*) and a valid PDO handle *ucPDOHandle* previously provided by *CMA_InstallPDO_E*.

CMA_Shutdown as well as a calling *CMA_Startup* again will remove all PDOs globally.

Function Call

short CMA_RemovePDO(

unsigned char ucChannelHandle
unsigned short usPDOHandle);

Parameter List

Parameter	Description
ucChannelHandle	CAN channel handle returned by *CMA_InitializeCard*.
usPDOHandle	PDO handle provided by *CMA_InstallPDO_E*.

Function Return Code

Return Code	Description
0:	Function completed successfully.
-103:	Function timeout.
-105:	*ucChannelHandle* not defined.
-113:	Client not active. Start-up required.
-126:	PDO handle not valid.

Related Events

None.

Example

PDOTransfer.c

See Also

4.4 PDO Transfer
4.4.1 PDO Buffer Configuration
6.15 CMA_InstallPDO_E

6.21 CMA_RequestGuarding

Description

CMA_RequestGuarding starts or stops the guarding / heartbeat supervision of the Slave node with Node-ID *ucNodeID*. Also the heartbeat production of the Client is controlled by this function. Controlling all nodes by a single function call is possible (Node-ID = 128).

The guarding / heartbeat parameters are individually configured by *CMA_AddNode*.

If the guarding fails the application is informed by event *GuardError* in the *Event-FIFO*.

Application Notes

CMA_RequestGuard requires an active Client (*CMA_Startup*) and a valid Node-ID introduced by *CMA_AddNode*.

Guarding and Heartbeat supervision of the Client itself is not possible. Only Heartbeat production of the Client can be controlled by this function.

Function Call

```
short CMA_RequestGuarding(
                unsigned char    ucChannelHandle ,
                unsigned char    ucNodeID,
                unsigned char    ucReqGuard);
```

Parameter List

Parameter	Description
ucChannelHandle	CAN channel handle returned by *CMA_InitializeCard*.
ucNodeID	[1-127]: Slave Node-ID or Client Node-ID for Heartbeat production 128: All nodes
ucReqGuard	Start/Stop flag. 0: Stop guarding / Heartbeat (supervision). others: Start guarding/ Heartbeat (supervision).

Function Return Code

Return Code	Description
0:	Function completed successfully.
-103:	Function timeout.
-105:	*ucChannelHandle* not defined.
-109:	*ucNodeID* not valid or not defined by *CMA_AddNode*.
-113:	Client not active. Start-up required.
-116:	Client can not be guarded

Related Events

Event	Description
GuardError	Guarding error occurred.

Example

NMTServices.c

See Also

4.6 Node Manager Configuration
4.6.2 Node Guarding
6.2 CMA_AddNode
6.19 CMA_RemoveNode
6.15 CMA_ReadEvent

6.22 CMA_SendRemotePDO

Description

CMA_SendRemotePDO sends a remote PDO request of the RPDO referenced by *usPDOHandle*. The replied answer of the Slave can be evaluated by the event *PDOReceived* in the *Event-FIFO*.

Application Notes

CMA_SendRemotePDO requires an active Client (*CMA_Startup*) and a valid PDO handle previously provided by *CMA_InstallPDO_E*.

Function Call

short CMA_SendRemotePDO (

unsigned char ucChannelHandle,
unsigned short usPDOHandle);

Parameter List

Parameter	Description
ucChannelHandle	CAN channel handle returned by *CMA_InitializeCard*.
usPDOHandle	PDO handle provided by *CMA_InstallPDO_E*.

Function Return Code

Return Code	Description
0:	Function completed successfully.
-103:	Function timeout.
-105:	*ucChannelHandle* not defined.
-126:	Invalid RPDO handle.

Related Events

None.

Example

PDOTransfer.c

See Also

4.4 PDO Transfer
4.4.2 PDO Services
6.15 CMA_InstallPDO_E
6.18 CMA_ReadPDO
6.20 CMA_RemovePDO
6.26 CMA_WritePDO
6.27 CMA_WritePDOBit

6.23 CMA_SetIntEvent

Description

CMA_SetIntEvent supplies the handle of an application defined WIN32 event to the CANopen Client API.

If the CANopen Client API has new asynchronous information available it informs the application by signalling the defined WIN32 event.

Application Notes

The function is only mandatory if the application is intended to evaluate the events in the *Event-FIFO* within a separate thread. Event usage and programming are described in section 5.2.

CMA_SetIntEvent can be called at any time after the successful initialization of the CAN channel by *CMA_InitializeCard*.

Function Call

short CMA_SetIntEvent(

unsigned char ucChannelHandle,
HANDLE hEvent);

Parameter List

Parameter	Description
ucChannelHandle	CAN channel handle returned by *CMA_InitializeCard*.
hEvent	WIN32 event handle

Function Return Code

Return Code	Description
0:	Function completed successfully.
-105:	*ucChannelHandle* not defined.

Related Events

None.

Example

Interrupt.c

See Also

4.2 Initialization
0 Event Processing
6.15 CMA_ReadEvent
6.8 CMA_InitializeCard

6.24 CMA_Shutdown

Description

Executing *CMA_Shutdown* the resources allocated by the CANopen Client API are released and the CAN controller is reset. The application is informed about the successful shutdown by the event *ShutdownCompleted*.

Application Notes

After calling *CMA_Shutdown* the parameters for bit timing, SDO transfer timeout and Node-ID are reset to their default values which can be redefined by *CMA_ConfigCANChannel, CMA_InitNodeMan* and *CMA_InitSDOMan*.

The function return code reports the success of the command transfer to the CANopen Client API. It does not confirm the success of the Client shutdown. This confirmation is reported by the event *ShutdownCompleted* in the *Event-FIFO.*

Function Call

short CMA_Shutdown(

unsigned char ucChannelHandle);

Parameter List

Parameter	Description
ucChannelHandle	CAN channel handle returned by *CMA_InitializeCard*.

Function Return Code

Return Code	Description
0:	Function completed successfully. Command posted to the Command FIFO.
-103:	Function timeout.
-105:	*ucChannelHandle* not defined.

Related Events

Event	Description
ShutdownCompleted	Shutdown completed successfully.

Example

```
// Setting baudrate to 1Mbit/s, no acceptance filter
CANConfig.ucBaudRate = 0;
ret = CMA_ConfigCANChannel (ucCANHandle, CANConfig);

// Configuration Node Manager with Node-ID=10
ret = CMA_InitNodeMan (ucCANHandle, 10);

// Configuration SDO manager with timeout = 1 s and a
// prolongation factor of  1
ret = CMA_InitSDOMan (ucCANHandle, 1000, 1);

// start-up Client  with new settings
ret = CMA_Startup (ucCANHandle);

        ...

// shutdown Client with new settings
ret = CMA_Shutdown (ucCANHandle);
```

See Also

4.2 Initialization
6.25 CMA_StartUp

6.25 CMA_Startup

Description

CMA_Startup initializes the CANopen Client by means of

Reset, initialization and start of the CAN controller

Initialization of the SDO and the Node Manager

Sending the Client boot up message
(1792 + Node-ID)

Change of Client state to PRE-OPERATIONAL

The successful start-up is reported to the application by the event *StartupCompleted*.

Bit timing, SDO transfer timeout and Node-ID of the Client are configured with the values defined beforehand by *CMA_ConfigCANChannel*, *CMA_InitSDOMan* and *CMA_InitNodeMan*. If these functions are omitted following **default settings** are used:

Data Link Layer:

Baudrate: 125 kbit/s
No acceptance filter.

Node Manager:

Client Node-ID=127

SDO Manager:

2s timeout of SDO response of a Slave.

After start-up the Client is in PRE-OPERATIONAL state, i.e.:

- Client is active on the bus.
- SDO transfers are possible.
- Guarding and SYNC may be configured and started.
- PDOs can be initialized by *CMA_InstallPDO_E*. PDO transfer is not possible.

For enabling the PDO transfer the Client must be switched into OPERATIONAL state by *CMA_ChangeState*.

A complete shutdown of the CANopen Client can be performed by *CMA_Shutdown*.

Application Notes

CMA_Startup can be called at any time after the successful initialization of the CAN channel by *CMA_InitializeCard*

The CANopen Client can be restarted by calling *CMA_Startup* again. In this case the CANopen Client clears the resources (PDO, SYNC) and restarts the CAN controller. The parameter settings of bit timing, SDO transfer timeout and Node-ID are preserved.

The function return code reports the success of the command transfer to the CANopen Client API. It does not confirm the success of the Client start-up. This confirmation is reported by the event *StartUpCompleted* in the *Event-FIFO.*

Function Call

short CMA_Startup(unsigned char ucChannelHandle);

Parameter List

Parameter	Description
ucChannelHandle	CAN channel handle returned by *CMA_InitializeCard*.

Function Return Code

Return Code	Description
0:	Function completed successfully. Command posted to the Command FIFO.
-105:	*ucChannelHandle* not defined.
-117:	Command FIFO is full.

Related Events

Event	Description
StartupCompleted	Start-up completed successfully.

Example

```
// Setting baudrate to 1Mbit/s, no acceptance filter
CANConfig.ucBaudRate = 0;
ret = CMA_ConfigCANChannel (ucCANHandle, CANConfig);

// Configuration Node Manager with Node-ID=10
ret = CMA_InitNodeMan (ucCANHandle, 10);

// Configuration SDO manager with timeout = 1 s and a
prolongation // factor of 1
ret = CMA_InitSDOMan (ucCANHandle, 1000, 1);

// start-up Client with new settings
ret = CMA_Startup (ucCANHandle);
```

See Also

4.2 Initialization
6.5 CMA_ConfigCANChannel
6.14 CMA_InitSDOMan
6.13 CMA_InitNodeMan
6.24 CMA_Shutdown

6.26 CMA_WritePDO

Description

CMA_WritePDO refreshes the data of a TPDO referenced by *ucPDOHandle*.

In case of an asynchronous TPDO the PDO is transmitted additionally if the Client is in OPERATIONAL state.

The PDO data can be refreshed partially determining the start byte position *ucDataOffset* and number of bytes *ucDataSize.* The new data are referenced by *ucpData*.

Application Notes

Writing the PDO data requires a valid PDO handle *ucPDOHandle* previously provided by *CMA_InstallPDO_E*.

Start byte position *ucDataOffset* and number of bytes *ucDataSize* must meet the following condition:

$$ucDataOffset + ucDataSize \leq 8$$

If the state of the Client is switched to OPERATIONAL all previously installed TPDOs are transmitted once automatically if initial transmission is not inhibited (see 6.15 CMA_InstallPDO_E).

Function Call

```
short CMA_WritePDO(
                unsigned char     ucChannelHandle ,
                unsigned short    usPDOHandle,
                unsigned char     ucDataOffset,
                unsigned char     ucDataSize,
                unsigned char     *ucpData);
```

Parameter List

Parameter	Description
ucChannelHandle	CAN channel handle returned by *CMA_InitializeCard*.
usPDOHandle	PDO handle provided by *CMA_InstallPDO_E*.
ucDataOffset	Byte position of the first byte of the TPDO data where the new data are written to. Range: [0..7] others: Invalid.
ucDataSize	Number of bytes of ucpData written to the TPDO data. Range: [0..7] with *ucDataOffset*+*ucDataSize* ≤ 8 others: Invalid.
ucpData	Pointer to the new PDO data.

Function Return Code

Return Code	Description
0:	Function completed successfully.
-103:	Function timeout.
-105:	*ucChannelHandle* not defined.
-113:	Client not started (*CMA_Startup*).
-126:	Invalid TPDO handle.
-127:	Invalid byte position (>7).
-128:	Invalid sum *ucDataOffset*+*ucDataSize*
-129:	Error accessing PDO

Related Events

None.

Example

PDOTransfer.c

See Also

4.4 PDO Transfer
6.15 CMA_InstallPDO_E
6.19 CMA_RemovePDO
6.22 CMA_SendRemotePDO
6.27 CMA_WritePDOBit
6.18 CMA_ReadPDO

6.27 CMA_WritePDOBit

Description

CMA_WritePDOBit sets a single bit at position *ucBitPosition* of the data of a TPDO to *ucBitValue.*

In case of an asynchronous TPDO the PDO is transmitted additionally if the Client is in OPERATIONAL state.

Application Notes

Writing the PDO data bit requires a valid PDO handle *ucPDOHandle* previously provided by *CMA_InstallPDO_E*.

If state of the Client is switched to OPERATIONAL all previously installed TPDOs are transmitted once automatically if initial transmission is not inhibited (see 6.15 CMA_InstallPDO_E).

Function Call

short CMA_WritePDOBit(

unsigned char	*ucChannelHandle ,*
unsigned short	*usPDOHandle,*
unsigned char	*ucBitPosition,*
unsigned char	*ucBitValue);*

Parameter List

Parameter	Description
ucChannelHandle	CAN channel handle returned by *CMA_InitializeCard*.
usPDOHandle	PDO handle provided by *CMA_InstallPDO_E.*
ucBitPosition	Bit position in the data of the TPDO where the new value is written to. Range: [0..63] others: Invalid.
ucBitValue	New bit value. 0: Bit value 0 others: Bit value 1

Function Return Code

Return Code	Description
0:	Function completed successfully.
-103:	Function timeout.
-105:	*ucChannelHandle* not defined.
-113:	Client not started (*CMA_Startup*).
-126:	Invalid TPDO handle.
-129:	Error accessing PDO
-130:	Invalid bit position.

Related Events

None.

Example

PDOTransfer.c

See Also

4.4 PDO Transfer
6.15 CMA_InstallPDO_E
6.19 CMA_RemovePDO
6.22 CMA_SendRemotePDO
6.26 CMA_WritePDO
6.18 CMA_ReadPDO

Appendix A

Table A-1: SDO error class and code

Error Class	Error Code	Example
(05) Service Error	(03) Parameter inconsistent	Toggle bit error
	(04) Illegal parameter	Timeout value reached.
(06) Access Error	(01) Access unsupported.	Writing a read-only or reading a write-only object.
	(02) Object not existent	Object not included in the dictionary.
	(04) Mapping error (should not happen, only for further extensions)	
	(06) Hardware fault	Hardware access error.
	(07) Type conflict	Data type does not match.
	(09) Attribute inconsistent	Sub-index does not exist.
(08) Other Error	(0)	Transfer aborted by the user.

Table A-2: Additional SDO error code

Additional Error Code		Example
00H in combination with:		
Error Class	Error Code	
5	3	Toggle bit error.
5	4	SDO protocol timed out.
6	1	Unsupported access to an object.
6	2	Object does not exist.
6	6	Access failed due to hardware error.
8	0	General error.
01H in combination with:		
Error Class	Error Code	
5	4	Client/Server command specifier not valid or unknown.
6	1	Attempt to read write only object.
02H in combination with:		
Error Class	Error Code	
5	4	Invalid block size.
6	1	Attempt to write read only object.
03H		Invalid sequence number.
04H		CRC error.
05H		Out of memory.
10H		Invalid service parameter.

11H	⇒ Sub-index not existent.
12H	⇒ Parameter length too large.
13H	⇒ Parameter length too small.
20H	Service cannot be executed.
21H	⇒ due to local control.
22H	⇒ due to current state.
30H	Value range exceeded
31H	⇒ Parameter value too large
32H	⇒ Parameter value too small
36H	⇒ Maximum value < minimum value
40H	Incompatible to other values
41H	⇒ Data cannot be mapped to PDO.
42H	⇒ PDO length exceeded.
43H	⇒ Parameter incompatibility.
47H	⇒ Parameter incompatibility in the device.

Glossary

API	**A**pplication **P**rogramming **I**nterface
CAN	**C**ontroller **A**rea **N**etwork
CiA	**C**AN **i**n **A**utomation
COB-ID	**C**ommunication **O**bject **I**dentifier
DIP	**D**ual-**I**nline **P**ackage
DPRAM	**D**ual-**P**ort **R**andom **A**ccess **M**emory
DS	**D**raft **S**tandard
ISO	**I**nternational **S**tandards **O**rganization
NMT	**N**etwork **M**anagement
OS	**O**perating **S**ystem
PC	**P**ersonal **C**omputer
RAM	**R**andom **A**ccess **M**emory
RPDO	**R**eceive **P**rocess **D**ata **O**bject
SDO	**S**ervice **D**ata **O**bject
TPDO	**T**ransmit **P**rocess **D**ata **O**bject

Index