User Manual

CANopen/CANopen-FD Master/Slave Protocol Stack

V 3.10.0

Version History

Version	Changes	Date	Editor	Release
1.0.2	Dynamic objects	2012/12/20	ged	
1.1.0	Change version to stack version	2013/03/09	boe	
1.2.0	Change version to stack version	2013/04/04	boe	
1.3.0	Sleep Mode added	2013/06/06	oe	
1.4.0	SDO block transfer added	2013/07/08	oe	
1.5.0	Object indication handling added	2013/10/02	oe	
1.6.0	Added new features	2014/09/05	ri	
1.7.0	Insert limit check	2014/09/05	ri	

Version	Changes	Date	Editor	Release
2.0.0	Add Multiline chapter	2014/11/15	boe	
2.2.0	Dynamic objects updated, network gateway	2015/05/15	ged	
2.2.4	Domain indication Bootup Procedure	2015/06/29	ged	
2.3.1	Split Indication/DynOd Application	2015/07/14	ged	
2.4.0	Add MPDO Usage	2015/08/25	ged	
2.4.3	Removed non CANopen msg	2015/10/29	phi	
2.6.1	Updated C#, LSS Slave, Store	2016/06/17	phi	
2.6.4	Add SDO client domain indication	2016/09/23	boe	
2.7.0	Adapt to library stack 2.7.0	2017/05/08	boe	
2.99.0	Added CAN-FD	2018/06/22	phi	
3.0.0	Release V3.0	2018/06/28	phi	
3.2.0	Domain/String-handling	2019/01/31	hil	
3.4.3	Updated Configuration Manager	2019/08/16	hil	
3.5.0	Change to emotas	2019/10/07	boe	
3.6.0	New version	2020/03/10	boe	
3.7.0	New version	2020/06/20	boe	
3.7.1	Add CANopen introduction	2020/08/21	boe	
3.7.4	Hints for Adaptation of cpu.c/h	2020/02/04	hil	
3.8.0	New version	2021/06/11	boe	
3.9.0	New version	2021/10/12	boe	
3.10.0	New version	2023/04/04	boe	

Table of Contents

Table of Images

References

CiA®-301	v4.2.0	Application layer and communication profile
CiA®-302	v4.1.0	Additional application layer functions
CiA®-303-3	v1.3.0	CANopen recommendation – Part 3: Indicator specification
CiA®-305	v2.2.14	Layer setting services (LSS) and Protocols
CiA®-401	v3.0.0	CANopen device profile for generic I/O modules

1 Overview

The CANopen/CANopen-FD Protocol Stack provides communication services for a CANopen/CANopen-FD compliant communication of devices and enables the fast and straight-forward integration of CANopen into devices. All services of CiA 301/1301 are supported (depending on version) by a user-friendly API. For simple portability to new hardware platforms the protocol stack is separated into a hardware-independent and hardware-dependent part with a defined interface.

Configuration and scaling is handled by a graphical configuration tool to generate optimized code and run-time efficiency.

2 Properties

- Support of CANopen Classic a CANopen FD
- Separation of hardware-dependent and hardware-independent part with defined interface
- ANSI-C compliant
- Compliance to mandatory MISRA-C rules
- support of all CiA 301/1301 services
- CiA-301 V4.2 or CiA-1301 V5.0 compliant
- configurable and scalable
- facility to add extension modules, e.g. for advanced master services
- flexible user interface
- static and dynamic object dictionary
- multiple expansion stages
- LED CiA-303

CANopen expansion stages

Service	Basic Slave	Master/Slave	Manager
SDO Server	2	128	128
SDO Client		128	128
SDO Transfer: expedited segmented block	 ● ● -	 ● ● ○	 ● ● ○
USDO Server	255	255	255
USDO Client		255	255
PDO Producer	32	512	512
PDO Consumer	32	512	512
PDO Mapping	Static	Static/dynamic	Static/dynamic
MPDO Destination Mode		○	○
MPDO Source Mode		○	○
SYNC Producer		●	●
SYNC Consumer	●	●	●
Time Producer		●	●
Time Consumer		●	●
Emergency Producer	●	●	●
Emergency Consumer		127	127
Guarding Master			●
Guarding Slave		●	●
Bootup Handling		●	●
Heartbeat Producer	●	●	●
Heartbeat Consumer	1	127	127
NMT Master		●	●
NMT Slave	●	●	●
LED CiA-303	●	●	●
LSS CiA-305		●	●
Sleep Mode CiA-454	●	●	●
Master Bootup CiA-302			●
Configuration Manager			●
Flying Master		○	○
Redundancy		○	○

Service	Basic Slave	Master/Slave	Manager
Safety (SRDO)	○	○	○
Multiline		○	○
CiA-401 (U8/INT16)	○	○	○
CiA-xxx	○	○	○

● - included, ○ - optional

3 CANopen Basics

The introduction into CANopen shall inform about the basics of CANopen, but it does not replace reading the CANopen specifications CiA 301 4.2 and other CiA specifications.

3.1 Introduction

The CAN based communication protocol **CANopen** is specified and maintained since 1994 by the CAN in Automation e.V. association. CANopen has been standardized internationally as European standard EN50325-5 and the **latest specification is CiA 301 in version 4.2.** CANopen specifies both communication mechanism and as well device functionality in various device profiles.

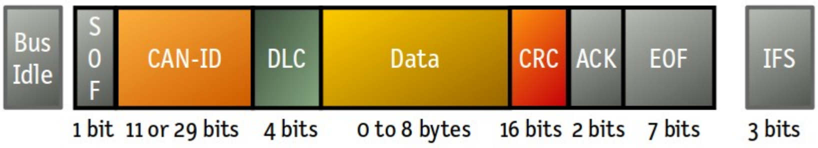
CANopen is used in various application fields such as:

- factory automation, process automation, production lines,
- embedded machine control,
- medical devices, operation room equipment, patient beds,
- HVAC control, lift control, door control,
- maritime application, off-shore and sub-sea applications,
- light rail vehicles, locomotives and passenger coaches,
- military applications, cranes, construction machines,
- light electric vehicles, truck superstructures, special purpose cars,
- ....

and many other application field with currently up to 65 different device profiles or application profiles addressing the various applications.

3.2 CAN the basis for CANopen

The CAN protocol according to ISO 11898 is the basis for CANopen. CANopen uses classical CAN messages with up to 8 data bytes (shorter message possible) and a bit rate of up to 1 Mbit/s.



CAN messages consist of

- a CAN-ID with 11 bit or 29 bit identifier
- a data length code (DLC)
- 0 .. 8 bytes of payload
- a CRC and additional check mechanisms

The CAN standard (ISO 11898) covers the 2 lower layers of the OSI[1] model of communication layers. Theses are the physical layer, which describes how voltage levels and timings form bits, and the data link layer that described how bits are combined into CAN messages.

The lower the CAN-ID, the higher the priority of the the CAN message.

The CANopen specification on top of CAN defines rules for the usage of CAN-IDs and payload to transfer application data, command and message to monitor the network in a defined way.

A CAN network usually consists of a trunk line with multiple short drop lines and a 120 ohm termination resistor at both ends.

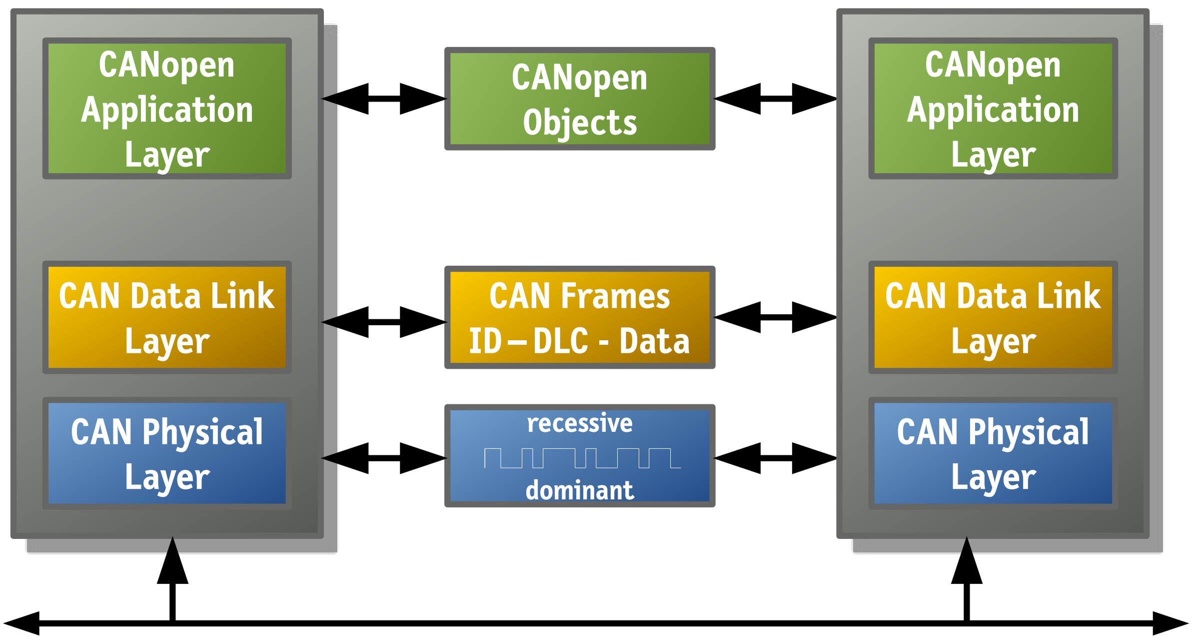
3.3 CAN-FD the basis for CANopen FD

CAN FD as the successor of CAN is used for **CANopen FD**. It is able to transfer up to 64 bytes of payload in a CAN FD message and the bit-rate in the data phase can be increased up to 8 Mbit/s. Anyway, CAN FD is not compatible with classical CAN.

Older CAN/CANopen devices may not be used together with CAN FD/CANopen FD-devices.

3.4 CANopen device model

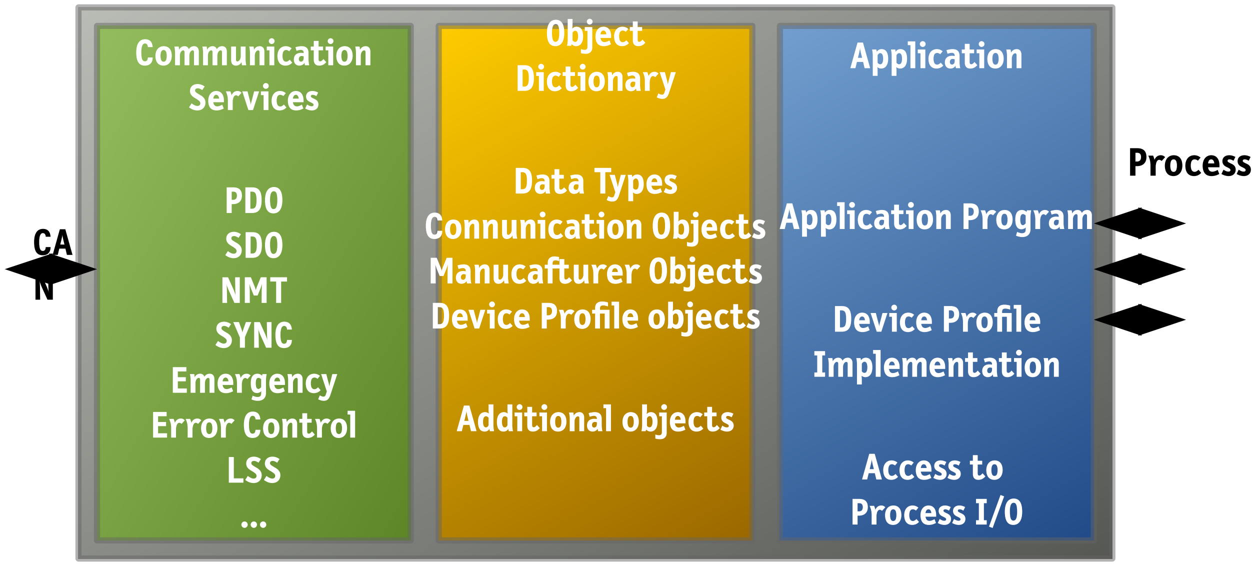
From a CANopen point of view CANopen devices exchange CANopen objects, that are mapped on CAN messages or – at the lowest layer - on some voltage levels on the physical layer. Such a CANopen object may be longer than 8 bytes, exceeding the length of a single CAN messages. Examples for such long objects are e.g. the device name or even a complete firmware of a device. The CANopen protocol ensures that these complete objects are transferred via CAN correctly.



1 Open Systems Interconnection model

CANopen devices consists of an application, an object dictionary and the CANopen communication services, that are combined in the CANopen stack. The application has access to the I/O ports of the devices and in most cases the application would also work without CANopen. But if this device shall exchange data with other devices in a CANopen network or shall be configured via CANopen or even updated via CANopen, it is required that CANopen communication services (usually as a CANopen stack) are integrated into the firm of the device. The so called object dictionary is the data interface between the CANopen stack and the application.

The object dictionary includes all data of the application that shall be transferred via CANopen.



There may be up to 127 CANopen devices in a CANopen network and CANopen devices have node-IDs between 1 and 127. Theses node-IDs can be fixed, configurable (e.g. via rotary switch or internal memory) or dynamically assigned from a master device.

3.5 Object dictionary (OD)

The object dictionary of a CANopen devices contains all objects of the device. Each object is addressed by an 16-bit index and a 8-bit sub index. **Each entry in the object dictionary can be access via CANopen (according to access rights) and from the application.** Using emotas' CANopen stack the entries in the object dictionary may be real C variables of the application or memory managed by the CANopen stack itself.

According to the 16-bit index the object dictionary is separated in various segments.

Index area	Objects
0x0000	reserved
0x0001 - 0x009F	Data types
0x00A0 - 0x0FFF	reserved
0x1000 - 0x1FFF	**Communication Profile Segment**
0x2000 - 0x5FFF	**Manufacturer Specific Profile Segment**
0x6000 - 0x9FFF	**Standardised Device Profile Area for up to 8 device profiles (up to 0x800 objects)**
0xA000 - 0xAFFF	Process image for CANopen-PLC (CiA 405)
0xB000 - 0xBFFF	Process image for CANopen-CANopen-Gateway
0xC000 - 0xFFFF	reserved

Data from up to 8 device profiles are located in the device profile segment (0x6000 - 0x9ff). Manufacturer-specific data can be stored in the manufacturer specific profile segment (0x2000 – 0x5fff) and the communication segment (0x1000 – 0x1fff) contains data to configure CANopen itself.

Each object may have different object codes and attributes:

Attribute	Description and value range
Object Code	Variable, Array, Record, Domain
Data Type	Unsigned8, Unsigned16, Real32, ..
Access	read only(ro), write only(wo), constant(const), read write(rw), *read write write(rww), read write read(rwr)*
Category	mandatory, optional, conditional
PDO Mapping	no, optional, default
Value Range	Value range of the object, check via SDO write access
Default Value	Default value after initialization

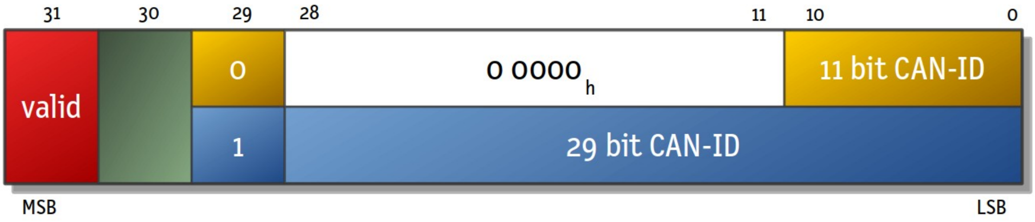
A CANopen object with the object code VARIABLE is similar to a C variable – a single value with a data type. An ARRAY is a structured data type with multiple elements of the same data type, just like an array in C and a RECORD is similar to a struct in C with multiple elements that may have different data types. A DOMAIN is just unstructured memory – for example the memory to store a new firmware.

Using CANopen one can use the services SDO, (USDO), PDO,MPDO and SRDO to acccess all or a sub set of the objects. From the application the access using the emotas CANopen stack is done either by type-safe and thread-safe API functions (e.g. coOdGetObj_i16) or directly to the application variable (e.g. gMyValues.TemperatureSensors[3]).

3.6 Communication objects (COB)

Communication objects (COB) are used in CANopen to exchange data and these communication objects are configured in the communication segment of the object dictionary. Confirmed services (e.g. SDO) use 2 communication objects with 2 different CAN-IDs and unconfirmed services (e.g. PDO, Emergency) use only one communication object with a CAN-ID to transfer data.

The COB-ID is an UNSIGNED32 parameter that contains the CAN-ID and 3 additional bits to configure the communication object.



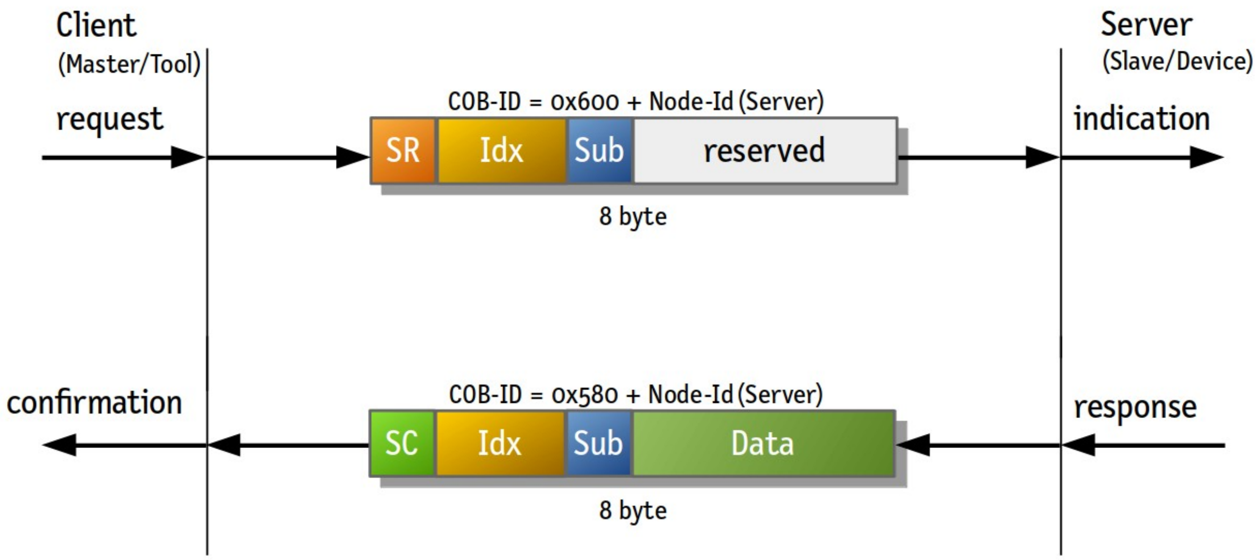
The MSB (31) defines mostly if a COB-ID is valid (0) or deactivated (1) and the 29[th] bit defines if a 11-bit CAN-ID shall be used or an 29-bit CAN-ID. The meaning of the bit 30 depends on the communication service.

The COB-IDs can be modified via CANopen or from the application. Using the emotas CANopen stack the API function coOdSetCobId() can be used to change the value of a COB-ID.

3.7 Service Data Object (SDO)

Service Data Objects (SDOs) provide arbitrary access to all objects of the object dictionary, but are used mostly to configure the device. A CANopen slave devices has one ore more SDO server and a CANopen Master additionally one or more SDO clients. A single SDO access can be used to read or write a single sub index of an object.

The SDO client initiates the SDO transfer to an SDO server and the SDO server respons with data from the local object dictionary of the SDO server. Writing to a device is called 'SDO Download' and reading from a CANopen device is called 'SDO Upload'. There are different types of SDO transfers to handle objects of various sizes. The expedited SDO transfer is able to transfer only 4 bytes of payload. The segmented SDO transfer can handle up to 4 GB of data and the SDO block transfer can transfer up to 4 GB even faster. The emotas CANopen Stack supports all SDO transfers with an identical API. The stack always picks the right SDO transfer depending on the size of the object and capabilities of the other communication partner.



If a CANopen slave only has one SDO server (like most devices), this SDO server uses the default COB-IDs for the 1st SDO server. These are

- 0x600 + Node-ID of the server for communication from client to server
- 0x580 + Node-ID of the server for communication from server to client

As each CAN-ID may only be used by one device, only one SDO client (master/tool) may access a single SDO server.

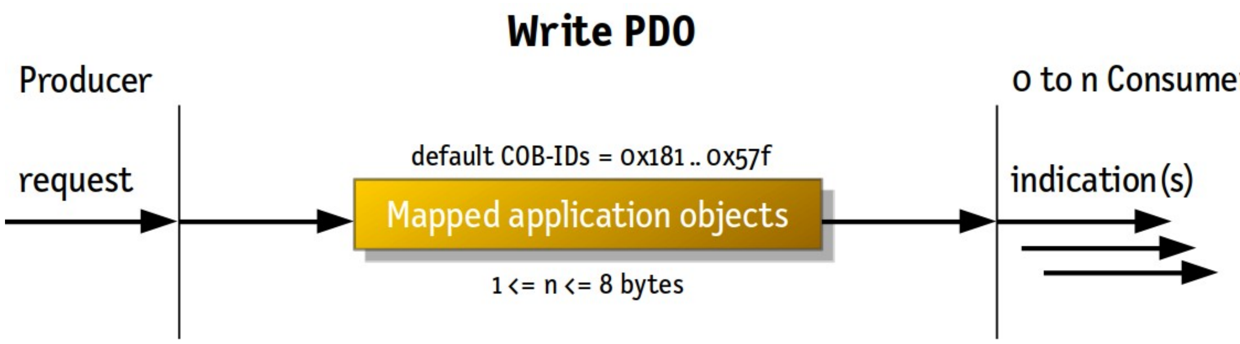
If simultaneous accesses to one device are required, multiple SDO servers can be implemented in one device. CiA e.V. does not define CAN-ID pairs for additional SDO connections, but the system integrator has to configure them according to CANopen network. The objects to configure the SDO server are 0x1200 .. 0x127f and for the SDO clients 0x1280 .. 0x12ff. Object 0x1200 which describes the mandatory 1st SDO server is optional. The SDO communication parameters in these objects have the following structure:

Sub-Index	Name	Datentyp
0	Highest sub-index supported	UNSIGNED8
1	COB-ID Client → Server	UNSIGNED32
2	COB-ID Server → Client	UNSIGNED32
3	Node-Id	UNSIGNED8

At all SDO accesses to objects in the object dictionary the access rights of the object are checked. For write access additionally the data type, the size and the value range is checked. SDO transfers can be terminated by both sides with a so called SDO abort message. The CANopen specification defines a long list of possible abort reasons the specific abort codes. The emotas CANopen stack aborts wrong SDO access automatically with a the right abort code (e.g. sub index does not exist). Additionally the application may send additionally abort codes like (0x06060000 access failed due to an hardware error).

3.8 Process Data Object (PDO)

Process data objects (PDO) are used to exchange process data with a high priority. For example with a CANopen battery one would transmit the current and voltage values by PDO, but a counter of operating hours or the manufacturer's name only by SDO. PDOs are transferred on CAN as CAN messages with a length of up to 8 bytes without any protocol overhead in the CAN messages. **Multiple sub indices of different objects can be transferred in one PDO.** The PDO communication is always broadcast from one PDO producer to one, many or none PDO consumers.



PDO can be sent synchronous or asynchronous. The transmission of synchronous PDOs is triggered by a SYNC message. The transmission of asynchronous PDOs is triggered by defined events. PDOs that are sent from a devices are called Transmit-PDOs (TPDOs) and PDOs that are received from a devices are called Receive-PDOs (RPDOs). Always the point of view of a specific devices is used. So the TPDOs of one device may be the RPDOs of another device.

A CANopen device may have up to 512 TPDOs and 512 RPDOs. In reality most devices, in particular CANopen slave devices, have less PDOs – often only 4 or a few more. All these PDOs must be configured in the communication segment in the object dictionary. Each PDO requires 2 objects for configuration. The PDO communication parameter objects defined the communication parameters (CAN-ID, transmission type, timings, ..) and the PDO mapping parameter objects define the content of a PDO.

Receive-PDOs (RPDO):

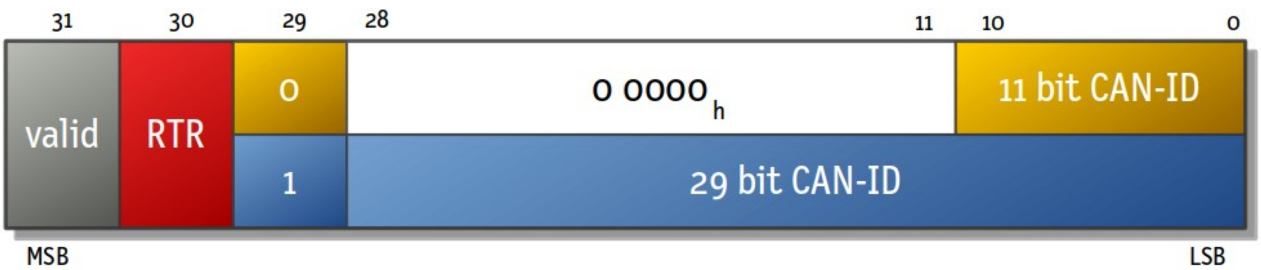
- communication parameters: index 0x1400 .. 0x15FF
- mapping parameters: index 0x1600 .. 0x17ff

Transmit-PDOs (TPDO):

- communication parameters: Index 0x1800 .. 0x19FF
- mapping parameters: Index 0x1a00 .. 0x1bff

The communication parameters included the COB-ID (sub index 1) with the CAN-ID, the transmission type (sub index 2), optionally an inhibit time (sub index 3), optionally an event timer (sub index 5) and optional a SYNC start value in sub index 6.

The PDO-COB-ID has the following structure:



The 3 most significant bits have the following meaning:

Bit	Values	Description
valid bit	0 1	PDO exists/is valid/processed PDO does not exist/is invalid/not precessed
RTR	0 1	PDO can be requested by RTR (only TPDOs) PDO cannot be requested by RTR (only TPDOs)
frame	0 1	11-bit CAN base frame format 29-bit CAN extended frame format

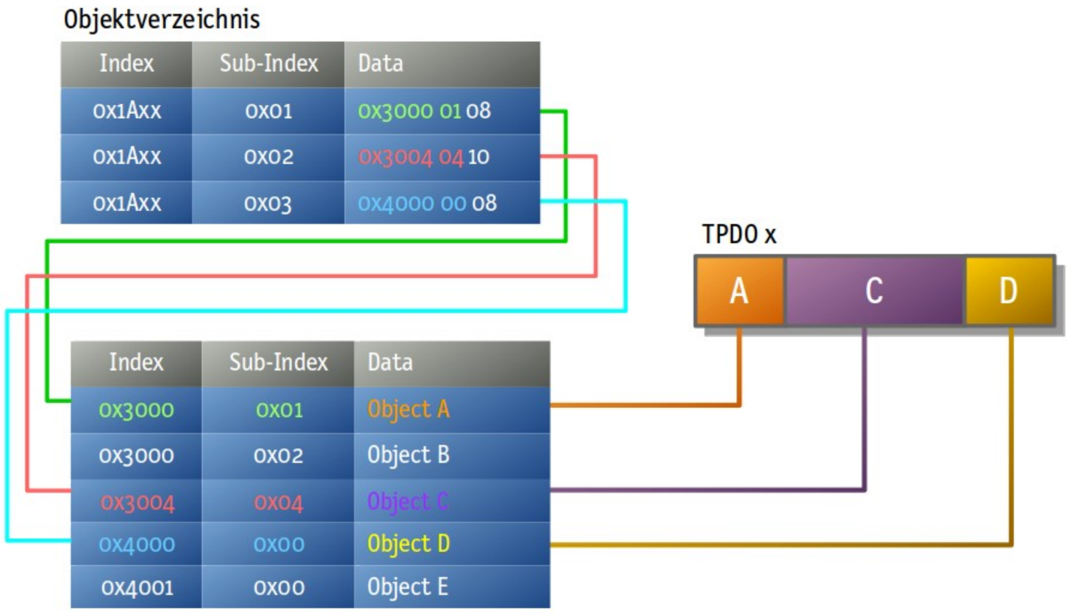
Possible values of the transmission type are:

Transmission Type	Description
0	Synchronous acyclic - transmission after next sync, only after changes
1 - 240	Synchronous cyclic - transmission after every 1st - 240th SYNC
241 - 251	reserved
252	Synchronous RTR only - only on request via RTR and SYNC
253	Asynchronous RTR only - only on request via RTR
254	Asynchronous (manufacturer specific) - asynchronous transmission
255	Asynchronous (device profile specific) - asynchronous transmission acc. to rules of profiles

The inhibit time prevents the transmission of event-triggered PDOs for a specific time and the event timer transmits an event-triggered PDO even without an event.

The PDO mapping in the PDO mapping tables (0x1600-0x7ff, 0x1a00-0x1bff) defines the assignment of CANopen objects from the object dictionary into PDOs.

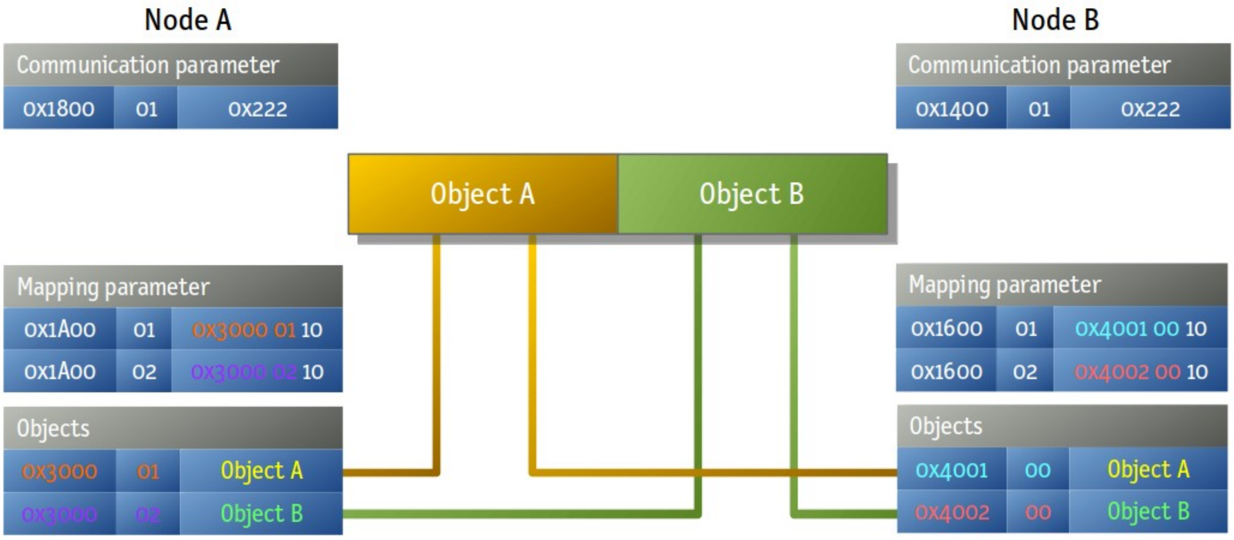
The mapping table (e.g. object 0x1a00 for TPDO 1) contains the objects, that shall be send in this PDO. In this example below sub index 01 contains a reference to object 0x3000, sub 1, length 8 bit and sub index 2 contains a reference to object 0x3004, sub 4, length 16 bit and sub index 3 contains a reference to object 0x4000, sub 0, length 8 bit.



If TPDO 1 shall be sent synchronously, the CANopen stack checks the mapping table when a SYNC message was received. If copies the current values of all referenced objects into the TPDO and transmits it.[2] In this example 3 objects with altogether 4 bytes are sent, but up to 8 bytes and be send/received in one PDO. If the mapping table of a PDO is defined at compile time, it is called static mapping. If the mapping table can be changed at run-time via SDO, it is called dynamic mapping. The emotas CANopen stack supports both variants.[3]

Connecting Transmit PDOs from one node with Receive PDOs from other nodes is called PDO linking. This PDO linking defines communication relationships between various produces an one or many consumers. If one device shall be able to receive a PDO from another one, at least the CAN-ID of both PDOs and the total length needs to match. Additional the mapped objects from both PDO mapping tables should correspond.

With most CANopen devices the configurable of the PDO linking is done by the system integrator or the CANopen master configures the PDO linking at start-up. With application profiles (e.g. CiA 417 'lift control', CiA 447 'car add-one devices' or CiA 454 'EnergyBus') the PDO linking is already defined.

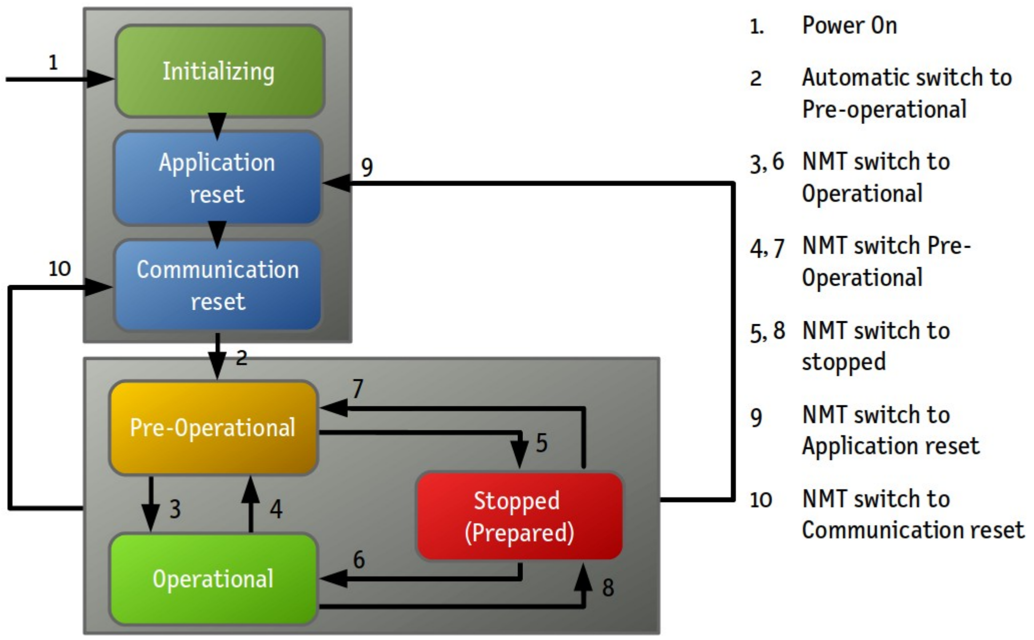


A PDO transfers data from objects of one device into objects of another device. So in all devices the data – which shall be transferred by PDO - need to be located in objects.

2 The real implementation is much more efficient. The stack already holds all points to these objects and just copies the data before transmission.
3 Static mapping is often much easier to handle for everybody.

3.9 CANopen State Machine

A CANopen devices has a state machine and some CANopen services can only be used in specific states.

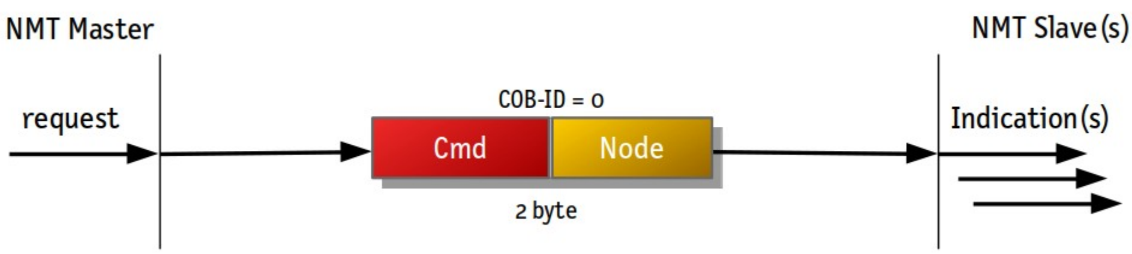


A CANopen device – that has a node-ID – automatically transits into Pre-Operational after the initialization. At the transition a CANopen Boot-Up message is sent from the devices, to indicate that it is ready to communicate. In the Pre-Operational state, all CANopen services but not PDO can be used. This Pre-Operational state is mostly used to configure the devices. Commands from the NMT master switch the state to Operational. In 'Operational' all CANopen services can be used and it is the normal operating state of a CANopen device. In the state 'Stopped' only NMT commands and Error Control messages are supported.

If the object 0x1029 is not implemented, CANopen devices automatically go from Operational into Pre-Operational in case of an error. Other autonomous state transitions are only allowed for self starting devices.

3.10 Network Management (NMT)

The CANopen network management services (NMT) services includes to control the CANopen state machine.



The NMT master in the CANopen network sends commands to individual or all CANopen slave devices in the ther CANopen to command state transitions.

The NMT command have the CAN-ID 0x00 and thus the highest priority in the CANopen network. The NMT messages consist of 2 data bytes.

The first byte is the command and the second byte is the target node-id. A node-ID 0 means that all devices are addressed.

The possible commands are:

- 1 – start node
- 2 – stop node
- 128 – enter pre-operational
- 129 – reset application
- 130 – reset communication

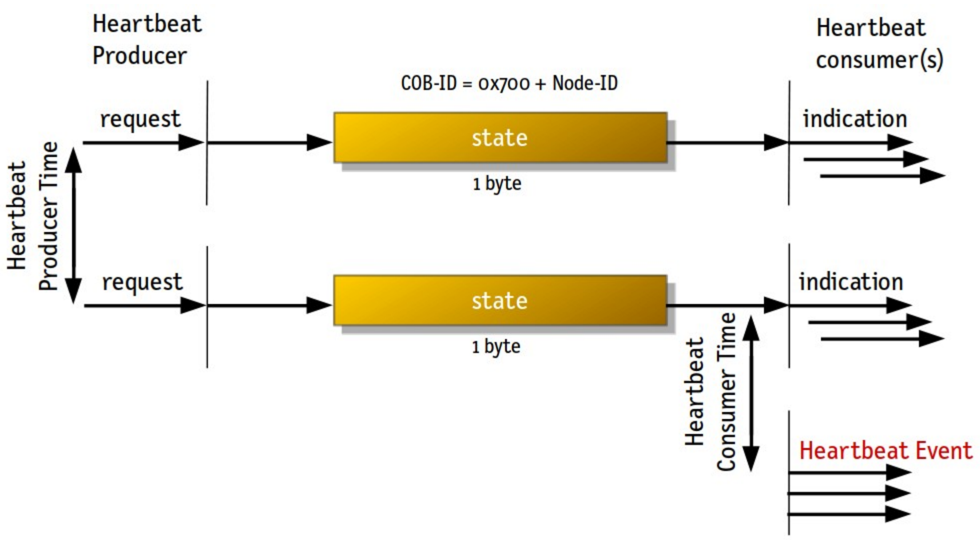
3.11 NMT Error Control (ErrCtrl)

NMT Error Control includes service, that can show and monitor a node's NMT state. An NMT master may monitor the state of a slave or even a communication loss but also a slave may monitor the master in the network. For example a drive could stop, if the master is not present anymore.

Bootup and Heartbeat messages are send with the CAN-ID 0x700 + node-ID of the devices and they only have 1 data byte. For boot-up messages the value of the data byte is always 0x00 and the heartbeat message the data bytes contains the current NMT state. Valid values are:

- *0 – bootup*
- 4 – stopped
- 5 – operational
- 127 – pre-operational

Heartbeat messages are sent cyclically. The cycle time of the heartbeat products can be configured in each device in object 0x1017.



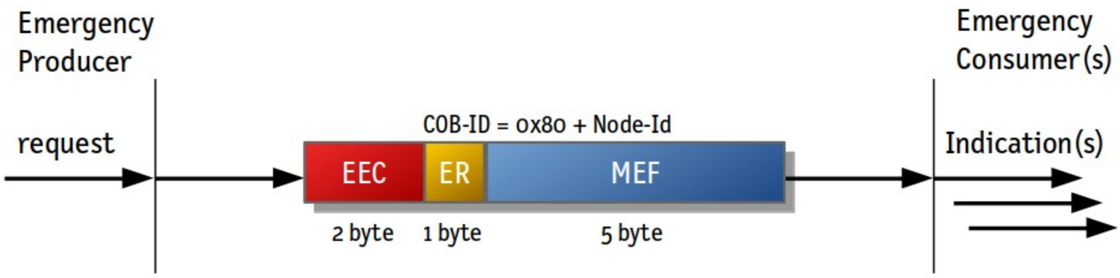
Heartbeat Consumers are configured in the object 0x1016. For example a devices with only have one heartbeat consumer sub index, if this device shall monitor only 1 device, or alternatively up to 127 heartbeat consumer entries.

The older Node Guarding and Life Guarding services are fully supported by the emotas CANopen Stack, but its usage is not recommend anymore for new devices. Both Node Guarding and Life Guarding use CAN-RTR messages, whose usage is not recommended anymore according to CiA AN802.

3.12 Emergency (EMCY)

CANopen Emergency (EMCY) messages are an optional part of the CANopen protocol to signal errors or warnings by CANopen devices. Nevertheless, we recommend to use Emergency messages. Devices, which send EMCY messages are called Emergency producer and devices, which receive EMCY messages are Emergency consumers. Most likely a CANopen slave is only an EMCY producer and CANopen master devices support the reception of EMCY messages. EMCY message are used to signal errors inside the devices, problems with CANopen configuration and CAN communication.

The CANopen specification CiA 301 defines a set a defined error codes and CANopen device profiles or CANopen application profiles may define additional error codes. Manufactures may use manufacturer-specific error codes as well and with each EMCY messages 8 manufacturer-specific bytes may be transmitted.



The length of an emergency message is always 8 bytes. The first 2 bytes contain the EMCY error code, the 3rd byte contains the value of the error register object (0x1001) and the last 5 byte contain manufacturer-specific data.

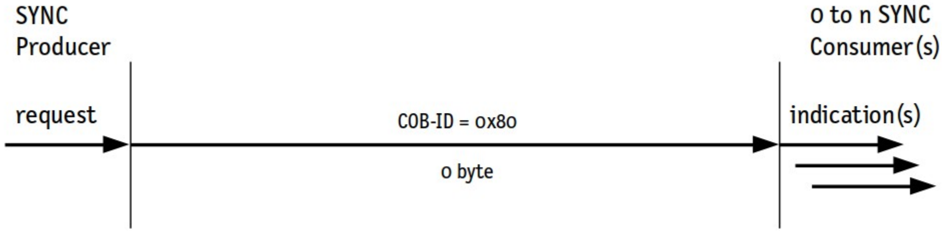
The CAN-ID according to the pre-defined connection set is 0x80 + node-ID of the producer.

The optional EMCY Inhibit Time object (0x1015) defines a minimal time between two EMCY messages, which can be used to prevent flooding the CAN with these messages.

Any EMCY producer may optionally implement the object 0x1003 with up to 254 sub indices. It the object is present, it is an error history which contains the error codes of all transmitted EMCY messages. The latest EMCY is always located at sub index 1.

3.13 Synchronization (SYNC)

A device in the CANopen network – usually the master device – may be SYNC producer and transmit cyclically SYNC messages with the CAN-ID 0x80 and the length of 0 or 1 bytes.



The SYNC messages are received by SYNC consumer and can be used to control the transmission of synchronous PDOs or to synchronize other actions. SYNC messages may optional contain a SYNC counter in the 1st byte and using the SYNC counter the PDO transmissions can be defined even more specific.

3.14 Predefined Connection Set

The Predefined Connection Set is a pre-definition for COB-IDs for specific CANopen services. According CiA 301 unconfigured CANopen devices need to follow the Predefined Connection Set.

Service	COB-ID	Configuration object
NMT	0x000	Not changeable
Bootup & Heartbeat	0x700+ Node-ID	Not changeable
SYNC	0x80	0x1005 (modification not recommended)
TIME	0x100	0x1012 (modification not recommended)
Emergency	0x80 + Node-ID	0x1014 (modification not recommended)
Default-SDO (Client→ Server)	0x600 + Node-ID (Server)	0x1200 (Not changeable)
Default-SDO (Server → Client)	0x580 Node-ID (Server)	0x1200 (Not changeable)
TPDO 1	0x180 + Node-ID	0x1800
TPDO 2	0x280 + Node-ID	0x1801
TPDO 3	0x380 + Node-ID	0x1802
TPDO 4	0x480 + Node-ID	0x1803
RPDO 1	0x200 + Node-ID	0x1400
RPDO 2	0x300 + Node-ID	0x1401
RPDO 3	0x400 + Node-ID	0x1402
RPDO 4	0x500 + Node-ID	0x1403

The emotas CANopen Stack sets the Predefined Connection Set automatically at start-up of the stack or after a 'reset communication'. A modification is possible using the load-indication-function of the stack.

CANopen Application Profiles such as CiA 417 'Lift Control', CiA 447 'Car Add-On Devices' or CiA 454 'EnergyBus' define different PDO COB-IDs!

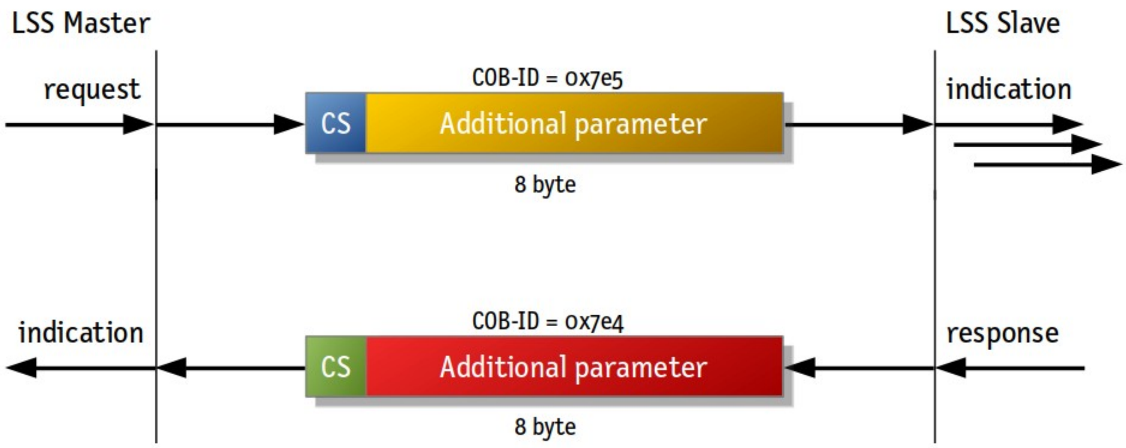
It is a common practice to reconfigure the COB-IDs of the PDOs, but one should not change the COB-IDs for SYNC, Time and Emergency.

3.15 Layer Setting Service (LSS)

Normally, CANopen device have a specific node-ID, but LSS as defined in CiA-305 defines a possibility to assign a node-ID dynamically. To address devices without a node-ID, the LSS master uses the so called LSS address to identify a device. The LSS address is a 128 bit value, which consists of the 4 sub indices of the identity object (0x1018): Vendor-ID, Product Code, Revision Number and Serial Number.

That means that all devices that shall support LSS, needs a serial number.

The LSS protocol only uses 2 CAN-IDs with a low priority. 0X7e5 for messages from the master to the slave and 0x7e4 for responses of the slaves. All different commands are distinguished by the the command specifier in the first data byte of the message.



The LSS sub services and procedures exceed the scope of this introductory chapter.

In general for new CANopen devices support of the LSS Fast Scan service is recommended.

LSS Fast Scan

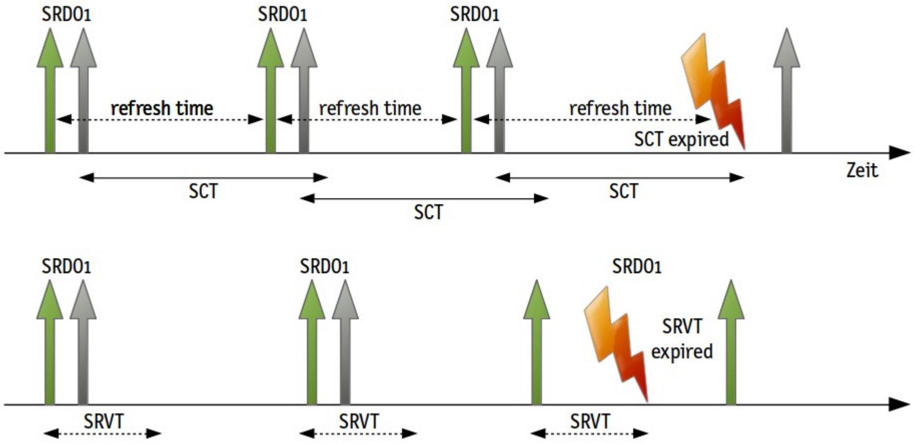
Using the LSS Fast Scan service even devices with unknown Vendor-ID and unknown product code can be identified. The emotas CANopen stack supports all LSS services.

3.16 Safety Relevant Data Object (SRDO)

SRDO is an optional CANopen service to transfer safety-relevant data according to SIL 3. The configuration and the communication itself is similar to PDOs, but there are additional properties. These are:

- cyclic date transfer with timeout monitoring (Safete Cylce Time)
- payload is transmitted twice, 2nd transfer is bit wise inverted
- data consistency between two CAN messages is checked.

- Time between normal and inverted CAN messages is checked (Safety Relevant Validation Time (SRVT))
- configuration is checked with a CRC to avoid unintended modification of the configuration



The SRDO functionality is a optional extension to the emotas CANopen Stack and it is descriped in a separate safety manual.

3.17 CANopen FD

The basic principles of CANopen and CANopen FD are the same. Anyway, there are the following differences:

- Length of PDOs exceed up to 64 bytes
- no RTR messages in CAN FD: no PDO requested by RTR, no Node-Guarding
- no bit wise PDO Mapping
- longer Emergency Messages with more information
- SDO replaced by more powerful USDO
- *June 2020: Additional services such as LSS or SRDO not yet defined for CANopen FD.*

The emotas CANopen FD stack supports selecting CANopen or CANopen FD at start-up of the device.

The USDO service in CANopen FD is an enhancement of the CANopen's SDO service. It is primarily intended to configure devices or transfer diagnostic or firmware data. Using CAN FD the CAN messages can be up to 64 bytes. The is a expedited USDO transfer, a segmented USDO transfer and a bulk transfer protocol. **Using USDO data can be transferred as unicast or broadcast.** For CANopen USDO transfer across CANopen network a specific Remote USDO service is defined.

4 CANopen Protocol Stack concept

- all services and functionalities can be switched on/off by `#define` directives
- configuration of the stack is done by the CANopen DeviceDesigner tool
- strict encapsulation of data, access only by function calls between different modules (no global variables)
- each service provides its own initialization function

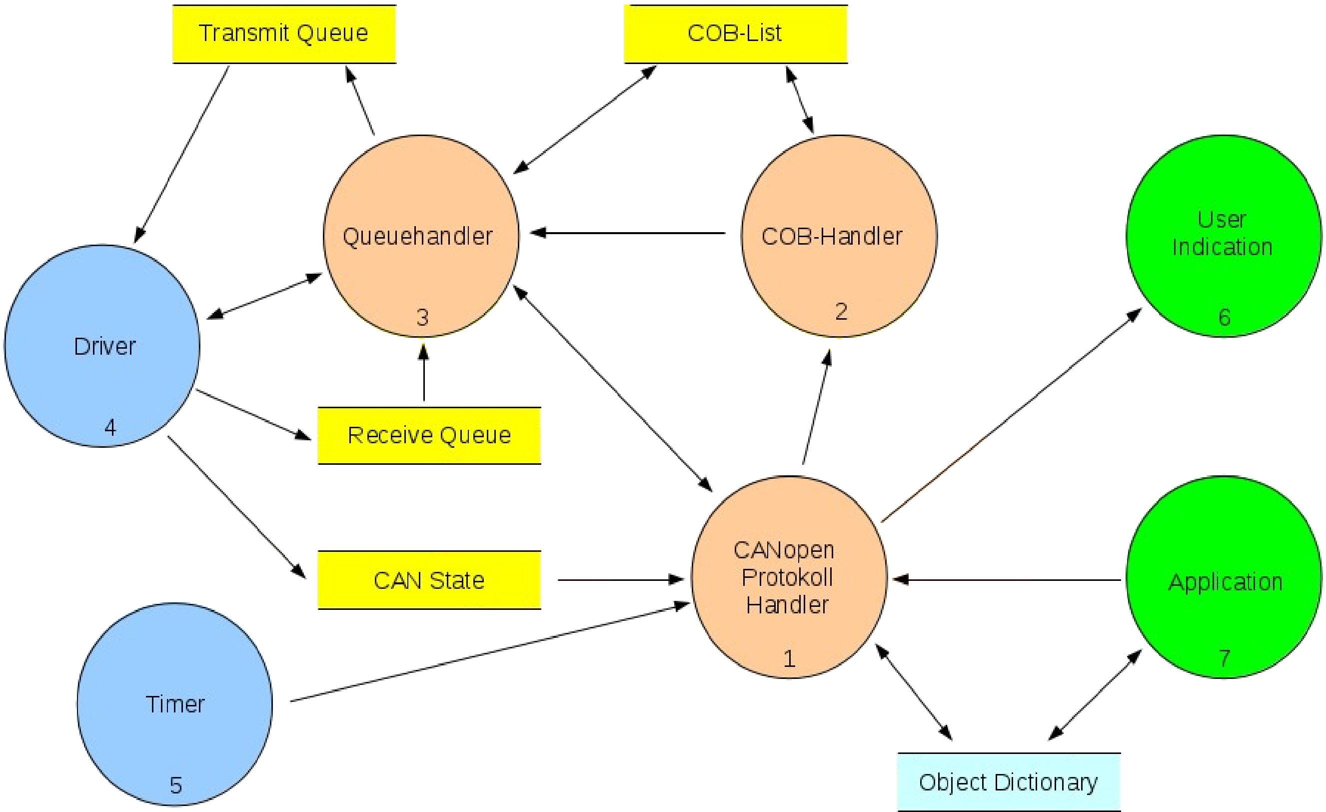


Illustration 1: Module Overview

The function blocks(FB)

- CANopen Protocol Handler (FB 1)
- COB Handler (FB 2)
- Queue Handler (FB 3)
- Driver (FB 4)

... are called by the central working function *coCommTask()*, in order to run all CANopen functions.

The central function has to be called if:

- new CAN messages are available in the receive queue
- the timer has expired
- the CAN communication state has changed.

If using an operating system, it can be indicated by signals. In embedded environments polling of the function coCommTask() is possible as well.

All function calls of CANopen service return the data type RET_T. If a function requests data from a remote node, the return value of the function is not the response but the state of the request. The response from the other node is signaled by an indication function, that has to be registered in advance(see chapter 4).

5 CANopen classic and CANopen FD

The User-Interface between CANopen and CANopen FD is identical and differs only in the following functions:

	CANopen classic (Single/Multiline)	CANopen FD (Single/Multiline)	CANopen classic + FD (Multiline)
SDO function/indication	present	-	present
USDO function/indication	-	present	present
EMCY Producer	3 parameters	6 parameters	6 parameters
EMCY Consumer	4 parameters	9 parameters	9 parameters

As long as the functionality of all CAN-Lines are identical (CANopen classic or CANopen FD), the respective function parameters apply. Using CANopen on one line, the CANopen FD function parameters apply for all lines, even if they are driven in CANopen classic mode. In this case additionally parameters for CANopen-FD are being ignored or handed over as 0.

If CANopen classic or CANopen FD principle is being supported, can be defined in the CANopen DeviceDesigner. The CANopen FD stack also provides the possibility to decide if the line is using CANopen classic or CANopen FD during the initialization. Therefore the function coCanOpenStackInit() awaits a list as parameter, which line is using CANopen classic or CANopen FD.

6 Indication Functions

The application can be informed about events or responses by the CANopen stack. The application must provide a function for each indication and register it at the stack. The registration can be done for each event type with the following function:

coEventRegister_<EVENT_TYPE>(&functionName);

It is possible to register multiple functions for an event. Then the function has to be called multiple times.
The maximal value has to be defined with the CANopen DeviceDesigner.
The data type for the functionName pointer depends on the CANopen service.

The following events can be registered:

EVENT_TYPE	Event	Parameters	Return value
COMM_EVENT	Communication state changed	Communication state	
CAN_STATE	CAN state changed	CAN state	
EMCY	automatically generated Emergency message shall be sent	Error Code Pointer to additional bytes	Send Emcy/Discard Emcy
EMCY_CONSUMER	Emergency Consumer message received	Node Id Error Code Error Register Additional Bytes	
LED_GREEN/LED_RED	Set red/green LED	On/off	
ERRCTRL	Heartbeat/Bootup State	Node Id HB State NMT Statue	
NMT	NMT State changed	new NMT state	Ok/not Ok
LSS	LSS slave information	Service bitrate pointer to ErrorCode pointer to ErrorSpec	Ok/not Ok
LSS_MASTER		Service-number ErrorCode ErrorSpec Pointer to Identity	
PDO	asynchronous PDO received	PDO Number	
PDO_SYNC	synchronous PDO received	PDO Number	
PDO_UPDATE	Update PDO data before transmission	Index Subindex	
PDO_REC_EVENT	Time Out for PDO	PDO number	

EVENT_TYPE	Event	Parameters	Return value
MPDO	Multiplexed PDO received	PDO number Index Subindex	
SDO_SERVER_READ	SDO Server Read Transfer begins	SDO server number index subindex	Ok/SDO abort code/Split indication
SDO_SERVER_WRITE	SDO Server Write Transfer finished	SDO server number index subindex	Ok/SDO abort code/Split indication
SDO_SERVER_CHECK_WRITE	SDO Server Check Write Transfer	SDO server number index subindex pointer to received data	Ok/SDO abort code
SDO_SERVER_DOMAIN_WRITE	SDO Domain size reached	Index subindex Domain Buffer Size Transfered Size	
SDO_CLIENT_READ	SDO Client Read Transfer finished	SDO client number index subindex number of data result	
SDO_CLIENT_WRITE	SDO Client Write Transfer finished	SDO client number index subindex result	
USDO_SERVER_READ	USDO Server Read Transfer begins	nodeId index subindex	Ok/SDO abort code/Split Indication
USDO_SERVER_WRITE	SDO Server Write Transfer finished	nodeId index subindex	Ok/SDO abort code/Split Indication
USDO_SERVER_CHECK_WRITE	SDO Server Write Transfer begins	nodeId index subindex pointer to received data	Ok/SDO abort code
USDO_SERVER_DOMAIN_WRITE	SDO Domain size reached	Index subindex Domain Buffer Size Transfered Size	
USDO_CLIENT_READ	SDO Client Read Transfer finished	nodeId index subindex result	

EVENT_TYPE	Event	Parameters	Return value
USDO_CLIENT_WRITE	SDO Client Write Transfer finished	nodeId index subindex result	
OBJECT_CHANGED	Object was changed by SDO or PDO access	index subindex	OK/SDO abort code
SYNC	SYNC message received		
SYNC_FINISHED	SYNC handling finished		
TIME	Time message received	Pointer to time structure	
LOAD_PARA	Restore saved objects	Subindex/OD segment	
SAVE_PARA	Store objects	Subindex/OD segment	
CLEAR_PARA	Delete stored values	Subindex/OD segment	
SLEEP	Sleep mode state	Sleep mode state	OK/Abort
CFG_MANAGER	DCF write finished	Transfer index subindex reason	
MANAGER_BOOTUP	Manager Event occurred	NodeID Event Type	
FLYMA	Flying Master state	State Master Node Priority	
SRD	SRD response from the Master	Result error code	
GW_SDOCLIENT_USER	Client SDO for Gateway functionality	-	SDO number

Each event can also be initialized by a static indication function at compile time. Static indication functions are always called after dynamic functions was executed.

All indication functions, that return a value, come with an additional argument:

Argument	Value	Meaning
execute	CO_FALSE	Test-mode – the function checks if the functionality can be executed with the given parameters. Return value of the function shall be evaluated Indication functionality may NOT be executed.
	CO_TRUE	Execution-mode – functionality will be executed with the given parameters. Return value of the function is not evaluated. Indication functionality shall be executed.

All registered functions are called with the argument execute = CO_FALSE. In this case the indication functions shall check, if the action shall be executed or not. Only if all functions request RET_OK, all indication functions are called again with execute = CO_TRUE in order to execute the corresponding actions.

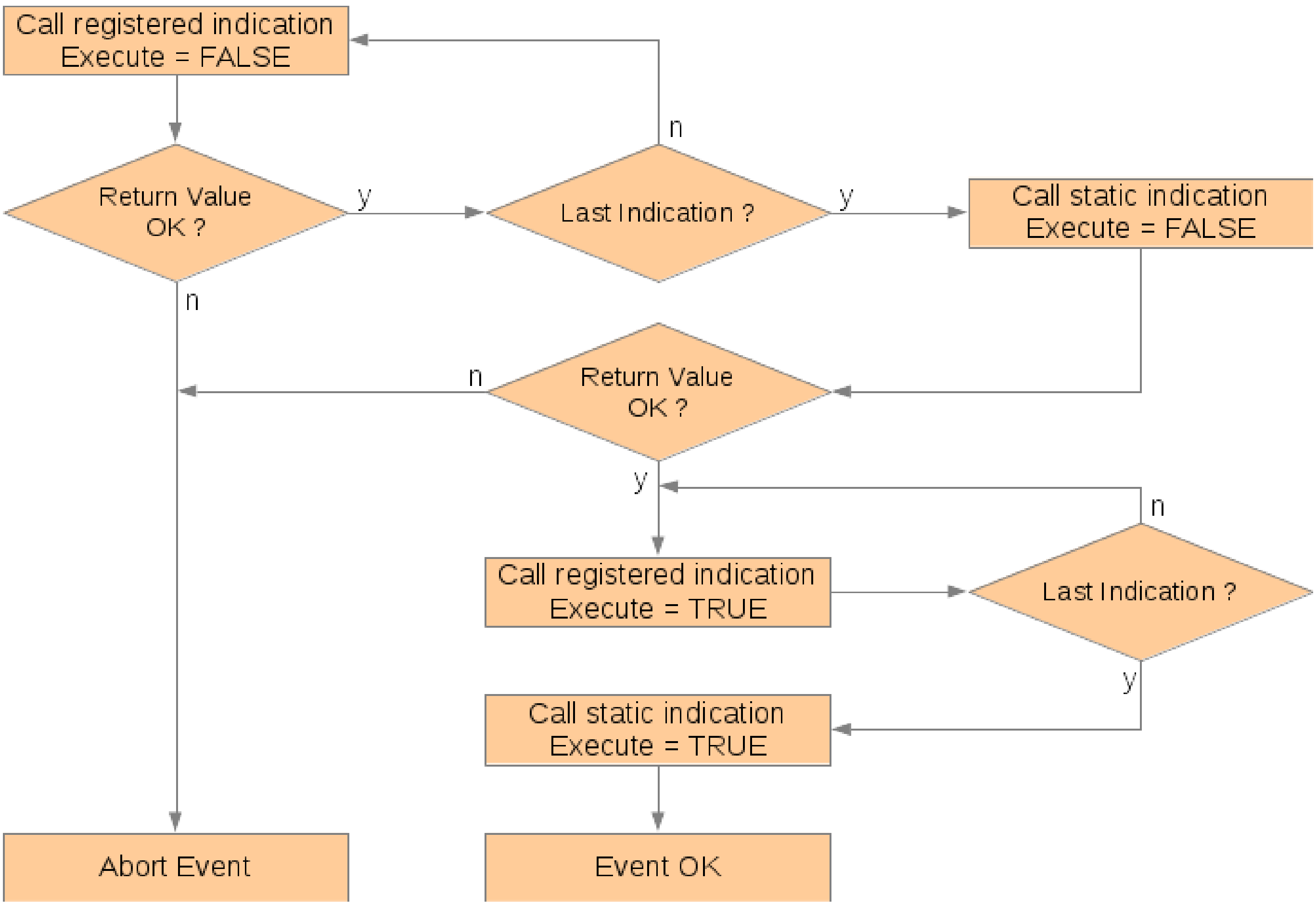


Illustration 2: Indications

7 The object dictionary

The object dictionary is generated by the CANopen DeviceDesigner and passed to the stack during the initialization. Gaps in subindices are allowed. All objects in the communication segment (1000h-1fffh) are managed by the corresponding service. The objects can only be accessed by function calls.

For all other objects there are 3 implementation options:

- managed variable (variable managed by stack)
- managed constant (constant managed by stack)
- pointer to variable in application-

For managed variables and constants there are access functions for the corresponding data types available: *oOdGetObj_xx* and *coOdPutObj_xx,* where xx is the data type of the object.
Additional attributes like access types, size information and default values can be retrieved using the functions *coOdGetObjAttribute()*, *coOdGetObjSize()* or *coOdGetDefaultVal_xx.*

The function *coOdSetCobid()* can be used to set COB-IDs of CANopen services.

The object dictionary implementation consists of 3 parts:

- variables (managed, constants, pointers)
- subindex descriptions
- object dictionary assignment of indices

7.1 Object dictionary variables

For each variable type up to 3 arrays can be created:

Managed variables:

```
U8      od_u8[] = { var1_u8, var2_u8 };
U16     od_u16[] = { var3_u16 };
U32     od_u32[] = { var4_u32, var5_uu32 };
```

Managed constants:

```
const U8      od_const_u8[] = { var6_u8, var7_u8 };
const U16     od_const_u16[] = { var8_u16 };
```

Pointer to variables:

```
const U8      *od_ptr_u8[] = { &usr_variable_u8 };
```

The definition and the handling of the arrays is done by the CANopen DeviceDesigner.

7.2 Object description

The object description exists for each sub index. It contains the following information:

Information	Meaning
subindex	Subindex
dType	Data type and implementation type (var, const, pointer, service)
tableIdx	Index in corresponding table
attr	Object attributes
defValIdx	Index in constant table for default value
limitMinIdx	Index in constant table for minimum value
limitMaxIdx	Index in constant table for maximum value

Definition of the attributes:

CO_ATTR_READ	Object is readable
CO_ATTR_WRITE	Object is writable
CO_ATTR_NUM	Object is a number
CO_ATTR_MAP_TR	Object can be mapped into a TPDO
CO_ATTR_MAP_REC	Object can be mapped into a RPDO
CO_ATTR_DEFVAL	Object has a default value
CO_ATTR_LIMIT	Object has limits
CO_ATTR_DYNOD	Object is dynamically created
CO_ATTR_STORE	Object shall be saved non-volatile
CO_ATTR_COMPACT	Object has identical subindices
CO_ATTR_FD	Object only applies in FD-Mode
CO_ATTR_STD	Object only applies in classical CAN Mode

The limit check for objects can be entered individually for each object using the CANopen DeviceDesigner.

7.3 Object dictionary assignment

The object dictionary assignment exists once for each index in the object dictionary. It consists of:

index	Index of the object
numberOfSubs	Number of sub indices
highestSub	Highest sub index
odType	Object Type (Variable, Array, Record)
odDescIdx	Index in object_description table

7.4 Strings and Domains

Strings are handled in 2 different ways:

- Constant strings are handled in the object dictionary. Therefore a list of pointers to the strings and a list of size information are implemented. Both lists are constant and cannot be modified.
- Variable strings have to be provides by the application. Pointers to these strings as well as the current and maximum length are handled in internal lists. To setup these settings, you can use coOdVisStringSet() and coOdSetObjSize() at run time. If you setup a default value for a variable string, the current and maximum length are set to the length of the default string.

Domains have to be provided from the application. Starting address, maximum size and current size have to be provided at run time, by using the functions coOdDomainAddrSet() and coOdSetObjSize().

When receiving a string or domain, the length of the object will be setup with the length of the received string or domain. When reading the object, the current length will be provided. Writing the object once again is always possible, but only up to the maximum length.

7.4.1 Domain Indication

Domains may have an arbitrary size and can also be used for program downloads. In this case they may not be stored completely in RAM, but have to be written to flash after a certain buffer size. The indication function coEventRegister_SDO_SERVER_DOMAIN_WRITE() may be used for this. The registered indication function is called after a defined number of CAN messages. The data may be written into flash and the corresponding domain buffer will be cleared and reused from beginning.

Attention! This behavior is applied to all domain objects. The specified size of CAN messages and reset of the buffer is done always when the size is reached. If other and larger domains shall be used, the data have to be copied to other buffers if necessary.

7.5 Dynamic Object Dictionary

7.5.1 Managed by Stack functions

Objects at the manufacturer and profile specific area can also be created dynamically at run-time. So it is possible to use already available or dynamic created variables from the application code with the object dictionary. These variables are linked to an object dictionary index and subindex. Dynamic objects can only be used with the following data types: NTEGER8, INTEGER16, INTEGER32, UNSIGNED8, UNSIGNED16 and UNSIGNED32.

To use dynamic objects dynamic memory is allocated by the stack at run time using malloc()[4]. This is realized using the function *coDynOdInit()* which needs to know the number of the dynamic objects. Objects itself are added using the function *coDynOdAddIndex()* and the sub-indices using *coDynOdAddSubIndex()*. These functions also specifies the attributes of the objects like access rights, PDO mapping information limits and more.

Dynamically created objects can be used with all functions which are provided by the CANopen stack and these objects can used in all services likes SDO or PDO without limitations.

Please refer to the example example_sl/dynod.

7.5.2 Managed by the application

Dynamic objects can also be created and managed by the application. To implement it, the application has to provide the following functions:

```
RET_T coDynOdGetObjDescPtr(                /* get Object description */
        UNSIGNED16 index,                  /* index */
        UNSIGNED8 subIndex,                /* subindex */
        CO_CONST CO_OBJECT_DESC_T **pDescPtr
UNSIGNED8 coDynOdGetObjAddr(               /* get address of object */
        CO_CONST CO_OBJECT_DESC_T    *pDesc /* pointer for description index */
UNSIGNED32 coDynOdGetObjSize(              /* get size of object */
        CO_CONST CO_OBJECT_DESC_T    *pDesc /* pointer for description index */
```

The stack always queries the object description and the size and the pointer after that. These objects may be used in PDOs as well. But the object must exist as long it is used, because the pointer is taken internally to refer to the mapped object. This is why the pointer to a dynamic object may not change as long as it is used in PDOs.

The example example_sl/dynod_appl may be used as a template for this functionality.

It is not possible to mix both functionalities.

4 malloc() is only used for dynamically created objects.

8 CANopen Protocol Stack Services

8.1 Initialization functions

Before using the CANopen Protocol stack, the following initialization functions have to be called:

coCanOpenStackInit()	Initialization of CANopen Stack and object dictionary
codrvCanInit()	Initialization of CAN Controllers
codrvTimerSetup()	Configuration of a time (e.g. hardware timer)
codrvCanEnable()	Start of CAN Controllers

8.1.1 Reset Communication

Reset of all communication variables (index 0x1000..0x1ffff) in the object dictionary to the default values. COB-IDs will be set according to the predefined connection set. At the end the registered event function (see coEventRegister_NMT())) is called.

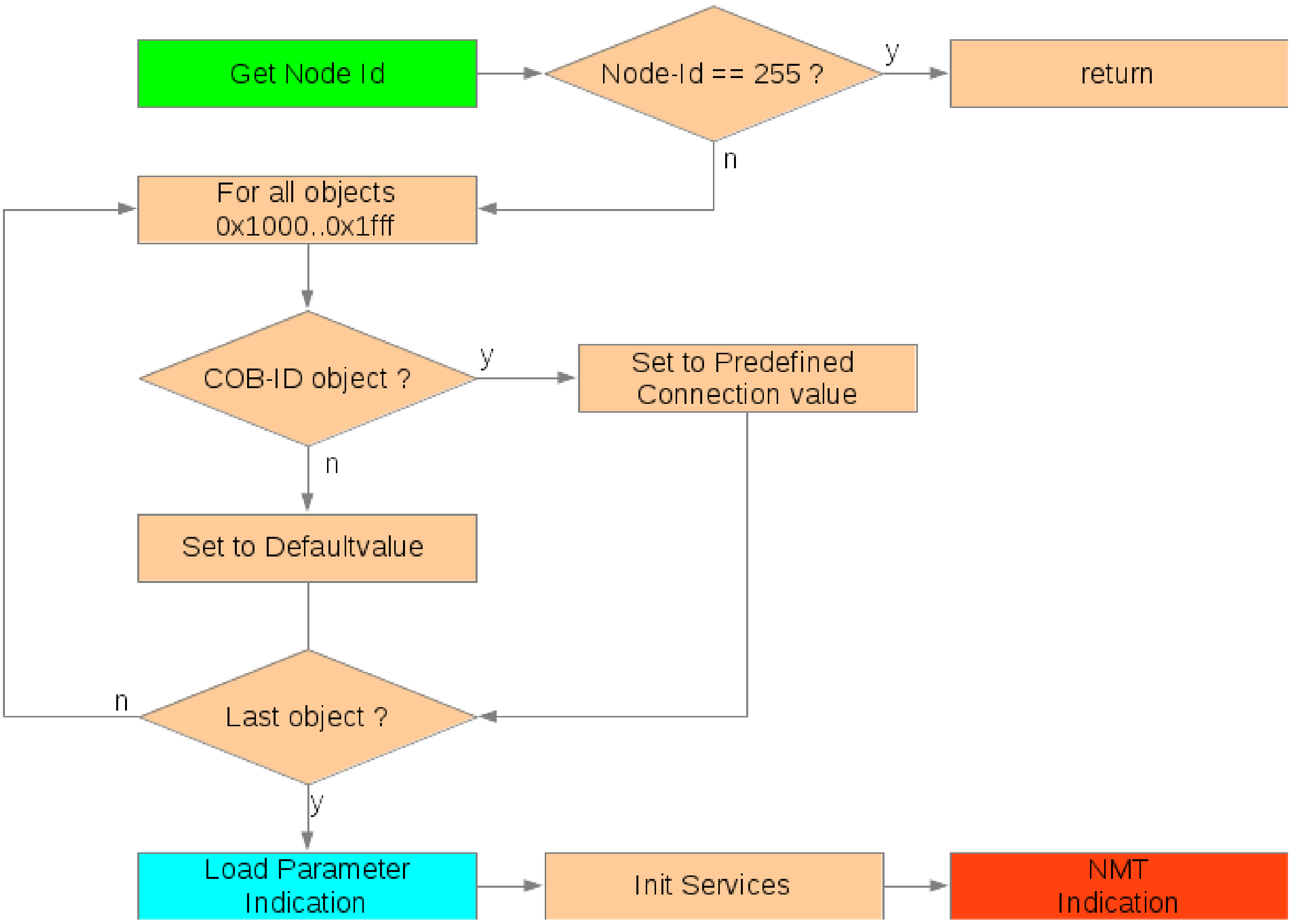


Illustration 3: Reset Communication

8.1.2 Reset Application

If an indication function is registered (see coEventRegister_NMT()), it can be called to do some actions in the application (e.g. to stop a motor). After that all object variables are reset to the default values and Reset Communication is executed.

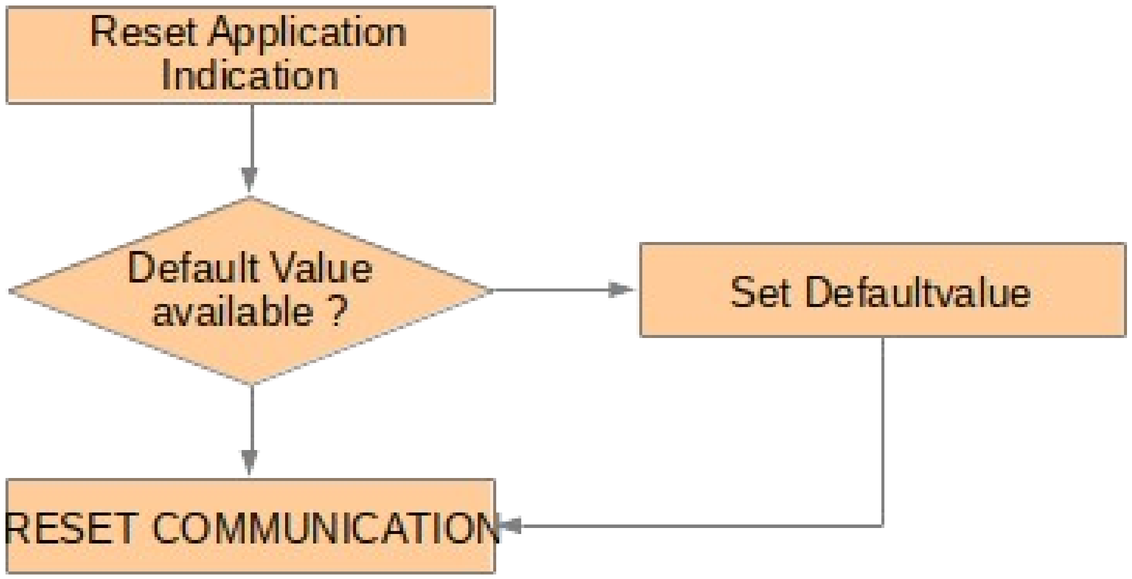


Illustration 4: Reset Application

8.1.3 Set node id

The node id have to be in a range of 1 to 127 or 255(data type unsigned char) and can be set via

- a constant at compile time
- a variable
- a function call
- LSS

This have to be entered in the input field at the CANopen DeviceDesigner.

Notes:

For LSS the node id must be set to 255u.

If the node id is provided via a function call or via a variable, the function prototype or the external variable declaration should be defined in gen_define.h

8.2 Store/Restore

The stack supports Store/Restore functionality only on request by writing to the objects 0x1010 and 0x1011. Reading the objects always returns the value 1.

The implementation of the non-volatile storage and restoring these values is part of the application.

8.2.1 Load Parameter

After Reset Communication or Reset Application the default values of the objects can be overwritten by the Load Parameter indication function. The function can be registered at the initialization of the CANopen stack using *coCanOpenStackInit().*

The indication function is called after each Reset Communication and Reset Application event and has to restore the parameters saved using Save Parameter (see 8.2.2).

It can also be used to set hard-coded values if the objects 0x1010 (store parameters) and 0x1011 (restore parameters) are not present.

8.2.2 Save Parameter

Saving of object values into non-volatile memory is done after writing the special value 'save' (0x65766173) into the object 0x1010. A corresponding function has to be registered using coEventRegister_SAVE_PARA() and this registered function shall handle the non-volatile memory storage. The selection of objects to be saved is application specific and can be defined within the registered function.

Which object can be stored is application specific. The CANopen static provides two functions, odGetObjStoreFlagCnt() and odGetObjStoreFlag(), to get the objects which are marked with the store flag by the CANopen Device Designer.

8.2.3 Clear Parameter

Deleting the values stored in non-volatile memory is done after writing the special value 'load' (0x64616f6c) into the object 0x1011. A corresponding function has to be registered using coEventRegister_CLEAR_PARA() and this registered function shall delete the content of the non-volatile memory. A following Reset Application or Reset Communication event shall not load any stored parameters when the Load Parameter Function (see 8.2.1) is called.

8.3 SDO

The COB-IDs of the first Server SDO are automatically set to the values defined in the Predefined Connection Set at Reset Communication. All other COB-IDs of SDOs are disabled after Reset Communication.

In general, COB-IDs can only be modified if the Disabled bit is set in the COB-ID in advance as required by the CANopen specification.

8.3.1 SDO Server

SDO Server services are passive. They are triggered by messages from external SDO Clients and react only according the received messages(request). The application can be informed about the start and the end of an SDO transfer by registered indication functions (see coEventRegister_SDO_SERVER_READ(), coEventRegister_SDO_SERVER_WRITE() and coEventRegister_SDO_SERVER_CHECK_WRITE()).

The SDO service evaluates the received data. It is checked if the objects are available in the object dictionary and if the access attributes are valid. After that the data are copied to or read from the object dictionary. Before and after the transmission indication functions can be called, which can modify the response of the server.

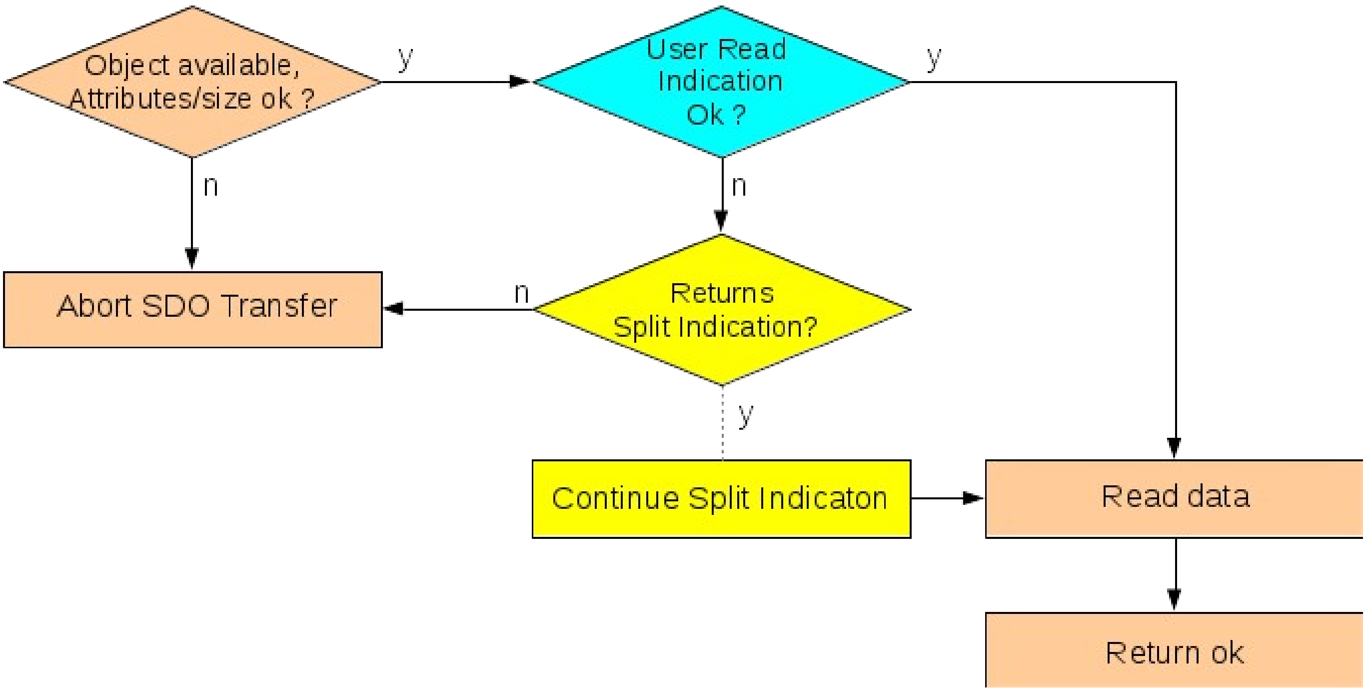


Illustration 5: SDO Server Read

The registered event functions may be left with the parameter RET_SDO_SPLIT_INDICATION. In this case the processing of the SDO request is stopped and the stack will not generate a response until the function coSdoServerReadIndCont() or coSdoServerWriteIndCont() is called. This mechanism can be used to read/write data from an external (e.g. I²C) component.

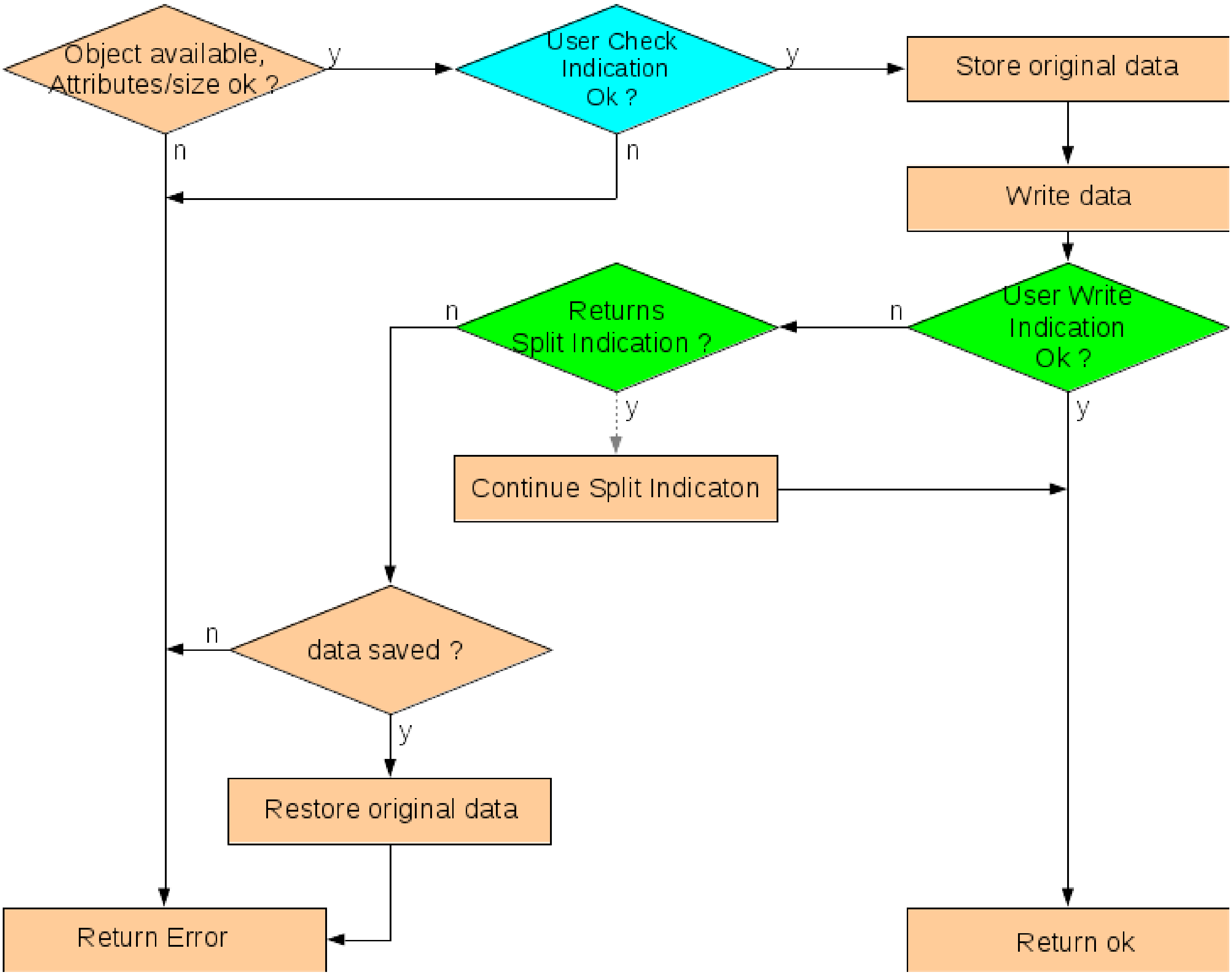


Illustration 6: SDO Server Write

8.3.2 SDO Client

SDO Client services must be requested (started) by the application. Reading a value from a remote device can be started with *coSdoRead()* and writing a value with *coSdoWrite().* Both functions start the SDO transfer. Later the application is informed about the result or an error by a registered indication function. (see coEventRegister_SDO_CLIENT_READ() and coEventRegister_SDO_CLIENT_WRITE()). For each SDO transfer a timeout is monitored, which aborts the transfer after the timeout. The configurable timeout value is valid for one CAN frame. If the transmission consist of multiple CAN frames (segmented transfer), the timeout restarts for each CAN frame.

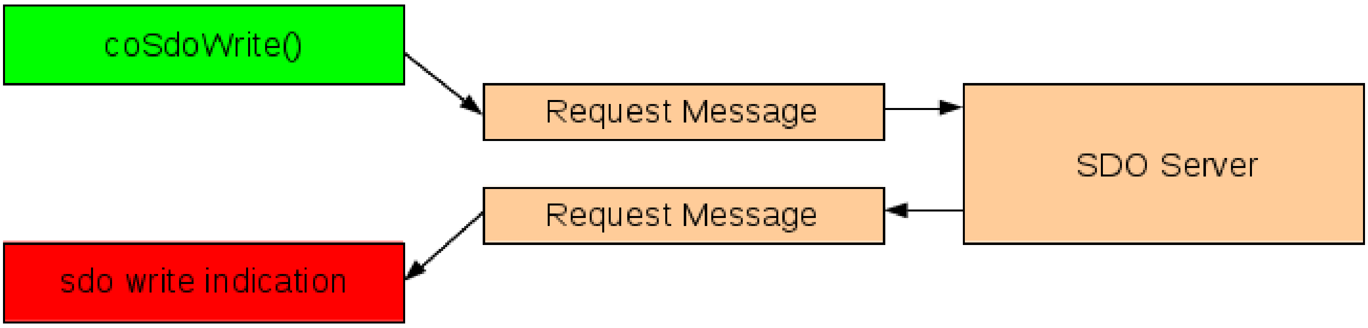


Illustration 7: SDO Client Write

8.3.3 SDO Block transfer

SDO Block transfer is automatically used by the SDO client as soon as the data size of the data to be transferred is larger than defined by the CANopen DeviceDesigner. Does the SDO server not support SDO Block transfer the client switches back to normal segmented transfer and repeats the request.
SDO requests to the server are always confirmed as SDO block transfer.
Calculation of the optional CRC within the Block transfer can be activated with the CANopen DeviceDesigner. Calculation itself is done using internal tables.

8.4 SDO Client Network Requests

SDO Client Network Requests are done by the functions *coSdoNetworkRead()* and *coSdoNetworkWrite()* *and are handled analog to the S*DO Client Read and Client Write calls.

8.5 USDO

USDOs are only available in CANopen FD mode. Using it at the same time with SDOs is not possible.

The COB-IDs for USDO are set according to the own Node-ID and are to changeable. Currently there are no configuration objects in the object dictionary necessary.

The configurations is done in the CANopen DeviceDesigner.

8.5.1 USDO Server

The USDO Server is a passive service. It is triggered by messages from external USDO Clients and react only according the received messages(request). The application can be informed about the start and the end of an USDO transfer by registered indication functions (see coEventRegister_USDO_SERVER_READ(), coEventRegister_USDO_SERVER_WRITE() and coEventRegister_USDO_SERVER_CHECK_WRITE()).

The number of simultaneous sessions can be configured in the CANopen DeviceDesigner.

The USDO Server service evaluates the received data. It is checked if the objects are available in the object dictionary and if the access attributes are valid. After that the data are copied to or read from the object dictionary. Before and after the transmission indication functions can be called, which can modify the response of the server.

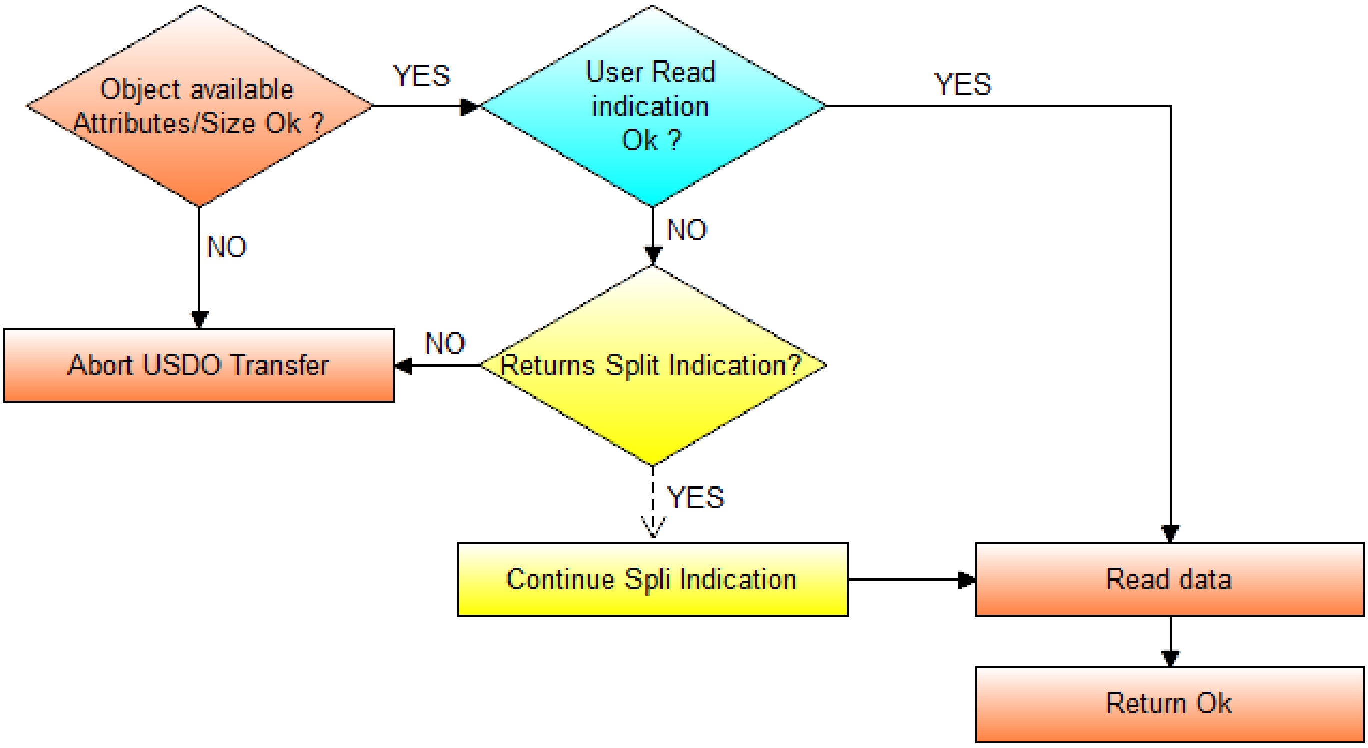


Illustration 8: USDO Server Read

The registered event functions may be left with the parameter RET_SDO_SPLIT_INDICATION. In this case the processing of the USDO request is stopped and the stack will not generate a response until the function coUsdoServerReadIndCont() or coUsdoServerWriteIndCont() is called. This mechanism can be used to read/write data from an external (e.g. I²C) component.

8.5.2 USDO Client

USDO Client services must be requested (started) by the application. Reading a value from a remote device can be started with *coUsdoRead()* and writing a value with *coUsdoWrite()*. Both functions start the USDO transfer. Later the application is informed about the result or an error by a registered indication function. (see coEventRegister_USDO_CLIENT_READ() and coEventRegister_USDO_CLIENT_WRITE()). For each USDO transfer a timeout is monitored, which aborts the transfer after the timeout. The configurable timeout value is valid for one CAN-FD frame. If the transmission consist of multiple CAN frames (segmented transfer), the timeout restarts for each CAN frame.

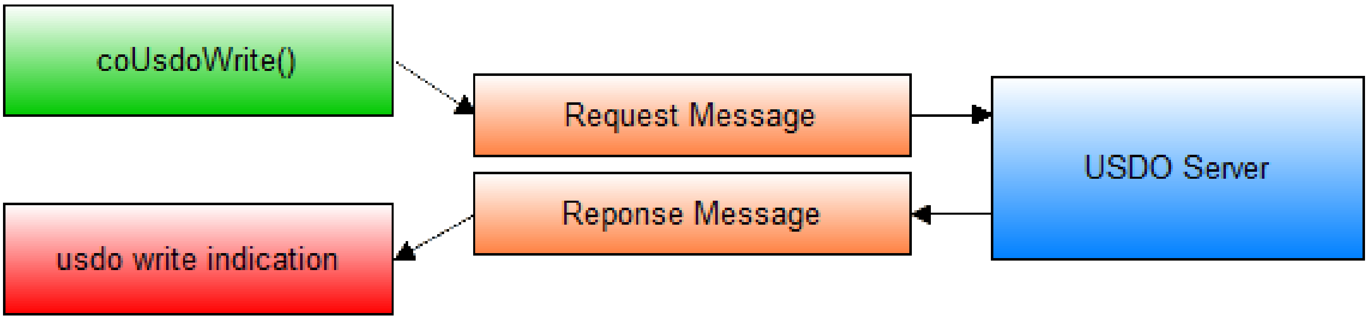


Illustration 9: USDO Client Write

8.6 PDO

PDO handling is done completely automatically by the CANopen stack. All data are copied from or to the object dictionary according to the configured PDO mapping. Inhibit time handling, timer-driven PDOs and synchronous PDOs are handled by the CANopen Stack as well.

If a PDO with a wrong length has been received and the Emergency service is enabled, the CANopen Stack sends an Emergency message automatically. The five application-specific bytes of the Emergency message can be modified in advance using an indication function (see coEventRegister_EMCY(). Without a modification by the application the Emergency message has the following content.

Byte 0..1	PDO Number
Byte 2..4	null

Synchronous PDOs are automatically handled. The receive PDOs data are copied to the objects at the reception of the SYNC message. For transmit PDOs the data are taken from the object dictionary and sent after the reception of a SYNC message.

The application can be informed about each received PDO by indication function. There are separate event indication functions for synchronous and asynchronous PDOs. (see coEventRegister_PDO() and coEventRegister_PDO_SYNC()).

8.6.1 PDO Request

Sending a PDO is only allowed for asynchronous and synchronous-acyclic PDOs. There are two functions available to send PDOs:

coPdoReqNr() Send PDO with defined PDO number

coPdoReqObj() Send PDO, which contains given object (index and subindex)

8.6.2 PDO Mapping

The PDO Mapping is made by mapping tables within the CANopen stack. For static mapping the constant tables are generated by the CANopen DeviceDesigner. For dynamic mapping the mapping tables are generated at the initialization of the stack or at the activation of a PDO mapping (writing to sub 0).

Structure of the mapping table:

```
typedef struct {
	void *pVar;		/* pointer to variable*/
	U8 len;		/* number of bytes to be sent */
	FLAG_T numeric;	/* flag to signal numerical values (for byte swapping) */
} PDO_MAP_ENTRY_T;
```

```
typedef struct {
        U8              mapCnt;                 /* number of mapped variables */
        PDO_MAP_ENTRY_T         mapEntry[];     /* Mapping entries */
} PDO_MAP_TABLE;
```

To change the PDO mapping of dynamic PDOs the following steps are required:

- disable the PDO (set NO_VALID_BIT in PDO COB-ID object)
- disable the mapping (set subindex 0 of mapping object to 0)
- modify the mapping objects
- enable the mapping (set subindex 0 to number of mapped objects)
- enable the PDO (reset NO_VALID_BIT in PDO COB-ID)

8.6.3 PDO Event Timer

The PDO Event Timer functionality can be used for asynchronous Transmit-PDOs and for all Receive-PDOs (not RTR). With Transmit-PDOs the PDO is sent automatically when the Event Timer has expired. With Receive-PDOs the timer is started at each reception of the PDO. If the timer expires, before a new PDO has been received, the application can be informed by a registered indication function. (see coEventRegister_PDO_REC_EVENT()).

8.6.4 PDO data update

PDOs are using data from the object dictionary to transmit them. If this data has to be updated before transmission, an indication function can be registered (see coEventRegister_PDO_UPDATE()).

8.6.5 RTR Handling

If the driver or hardware can not handle RTRs, bit 30 have to set at all PDO COB-Ids (0x4000 0000). With the define CO_RTR_NOT_SUPPORTED resetting this bit is prevented.

8.6.6 PDO and SYNC

The SYNC service allows to synchronize the data transmission and the data collection in the network. After the SYNC message has been sent all transmit PDOs are sent with the data from the object directory and all receive PDOs are entered to the object dictionary.

The data can be updated or retrieved from the object dictionary via the registered indication functions.

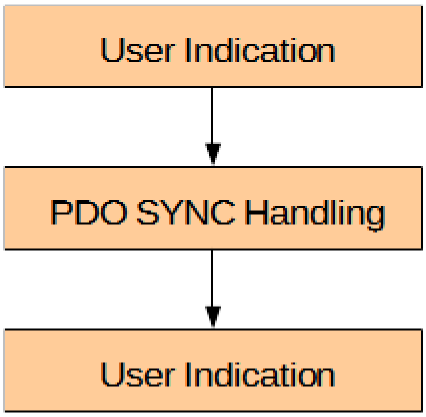


Illustration 10: PDO Sync

8.6.7 Multiplexed PDOs (MPDOs)

If the normal PDOs are not sufficient, a special kind of the PDOs the multiplexed PDOs may be used. These MPDOs do not contain a fixed mapping, but the index and subindex information of the data are transferred by MPDOs as well. In contrast to normal PDOs only one application object may be transferred by an MPDO.

Using the function register_MPDO() a callback function may be registered that is called when a MPDO is received. The transmission of MPDOs is done by coMPdoReq().

Hint: MPDOs can only be sent asynchronously and need to have the transmission type 254 or 255.

8.6.7.1 MPDO Destination Address Mode (DAM)

Using the Destination Address Mode the consumer information in which object the data shall be stored are transmitted by the producer:

Dst. Node	Dst. Index	Dst. Sub	Data

8.6.7.1.1 MPDO DAM Producer

Entries in Object Dictionary

Index	Sub-index	Description	Value
$18xx_h$		PDO Communication Parameter	
$1Axx_h$	0	Number of mapping entries	255
$1Axx_h$	1	Mapping entree	Appl.

8.6.7.1.2 MPDO DAM Consumer

Entries in Object Dictionary

Index	Sub-Index	Description	Value
$14xx_h$		PDO Communication Parameter	
$16xx_h$	0	Number of Mapping Entries	255

The received data as stored in the consumer according to the transmitted index/subindex

8.6.7.2 MPDO Source Address Mode (SAM)

Using the source address mode the producer information (source node, source index and source sub index) are transmitted in the MPDO.

Src Node	Src Index	Src Sub	Data

8.6.7.2.1 MPDO SAM Producer

The SAM Producer uses an Object Scanner List the contains all objects that may be sent by the MPDO. A device may only contain 1 MPDO in SAM Producer Mode.

Entries in Object Dictionary

Index	Sub-Index	Description	Values
$18xx_h$		PDO Communication Parameter	
$18xx_h$	2	Transmission Type	254/255
$1Axx_h$	0	Number of Mapping Entries	254
$1FA0_h$..$1FCF_h$	0-254	Scanner !!br0ken!!	

The format of the scanner list is:

MSB		LSB
Bit 31..24	Bit 23..8	Bit 7..0
Block Size	Index	Sub Index

8.6.7.2.2 MPDO SAM Consumer

Entries in Object Dictionary

Index	Sub-index	Description	Value
$14xx_h$		PDO Communication Parameter	
$16xx_h$	0	Number of Mapping Entries	254
$1FD0_h$..$1FFF_h$	0-254	Dispatcher List	

The dispatcher list is a cross reference between the producer object and the consumer object. Its format is:

Dispatcher list:

MSB					LSB
63..56	55..40	39..32	31..16	15..8	7..0
Block size	Local Index	Local SubIdx	Prod. Index	Prod SubIdx	Prod Node

Using the block size multiple identical sub indices may be described by one entry.

8.7 Emergency

8.7.1 Emergency Producer

Transmission of Emergency message can be triggered by the application or they can also be sent automatically at certain error conditions (CAN Bus-Off, wrong PDO length, …). Automatically sent PDOs can be modified by the application as well and its transmission can even be prohibited by the application by a registered indication function (see coEventRegister_EMCY()).

8.7.2 Emergency Consumer

Emergency Consumers are configured by writing the COB-IDs into the object 0x1028 in the object dictionary. All COB-IDs in the object 01028 are received and interpreted as emergency messages. The application is informed about the reception of each emergency message by a registered indication function. (see coEventRegister_EMCY_CONSUMER()).

8.8 NMT

NMT state changes are usually initiated by the NMT Master who sends the NMT commands that has to be executed by all NMT slaves. The only exception is the transition to OPERATIONAL, which can be rejected by the application. For this case a registered indication function is called (see coEventRegister_NMT()). With the return value of this function the application can decide if the transition to OPERATIONAL is possible.

In certain situations the application may change the NMT state from OPERATIONAL to PRE-OPERATIONAL or STOPPED. These situations may be error conditions like loss of heartbeat or CAN bus-OFF. The reaction on these events are defined in the object 0x1029, which is evaluated by the CANopen stack.

8.8.1 NMT Slave

NMT slave devices react on the NMT commands sent by the NMT master. The application can be informed about NMT state changes by a registered indication function. (see coEventRegister_NMT()).

8.8.2 NMT Master

The NMT master can change the NMT state of all nodes in the network by the function coNmtStateReq(). The NMT command can be sent to individual nodes or to the complete network(0). For the latter case an additional parameter defines, if the command is also valid for the own master node.

8.8.3 Default Error Behavior

The default error behavior (Heartbeat consumer event or CAN bus off) can be defined in the object 0x1029. If the object does not exist, the node automatically switches into the NMT state PRE-OPERATIONAL at these errors. If the emergency producer is activated, an emergency message is sent automatically. If an emerge indication function is registered, the content of the 5 additional bytes of the emergency messages can be modified. (see coEventRegister_EMCY()).

8.9 SYNC

The transmission of the SYNC message is started, if the SYNC producer bit is set in the object 0x1005 and if the SYNC interval in object 0x1006 is greater than 0. There are 2 possible indication functions for SYNC handling (see coEventRegister_SYNC() and coEventRegister_SYNC_FINISHED()):

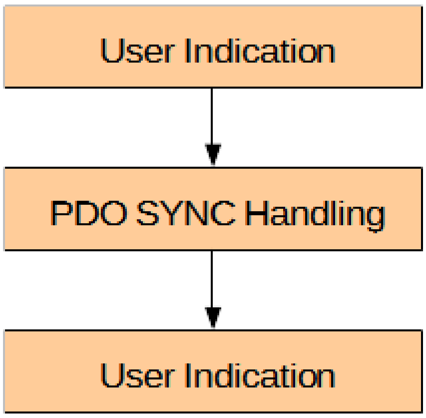


Illustration 11: SYNC Handling

8.10 Heartbeat

8.10.1 Heartbeat Producer

If a new heartbeat producer time is set in object 0x1017, it is immediately used by the CANopen stack. At the same time the first heartbeat message is sent if the value is unlike 0.

8.10.2 Heartbeat Consumer

The configuration of Heartbeat consumers can be done by the function *coHbConsumerSet()* or by writing to the corresponding objects 0x1016:1..n in the object dictionary.

If the function *coHbConsumerSet()* is used, the Heartbeat consumer is automatically configured in the object 0x1016 if there is a free entry available. Otherwise an error is returned. Bootup messages are received by all nodes, even if the heartbeat consumer is not configured for the remote nodes.

If a monitoring state is changed, a registered indication function (see coEventRegister_ERRCTRL()) is called. The possible state changes are:

CO_ERRCTRL_BOOTUP	Bootup message received
CO_ERRCTRL_NEW_STATE	NMT State changed
CO_ERRCTRL_HB_STARTED	Heartbeat started

CO_ERRCTRL_HB_FAILED	Heartbeat lost
CO_ERRCTRL_GUARD_FAILED	Guarding from master lost
CO_ERRCTRL_MGUARD_TOGGLE	Toggle error of the Slave
CO_ERRCTRL_MGUARD_FAILED	Guarding of the slave lost
CO_ERRCTRL_BOOTUP_FAILURE	Bootup transmission error

8.11 Life Guarding

Life Guarding is automatically activated if the values of the objects 0x100c and 0x100d are unlike 0 and the first Guarding message from the master has been received. When the configured guarding time resp. the life time factor has expired, the standard error behavior is executed (see chapter 8.8.3Default Error Behavior) end a registered indication function is called (see coEventRegister_ERRCTRL()).

8.12 Time

The time service can be used as producer or consumer. At the initialization it has to be defined if it shall be a Time producer or consumer. To send time message the function *coTimeWriteReq()* can be used. Incoming time messages are signaled by a registered indication function (see coEventRegister_TIME()).

8.13 LED

For LED signaling according to CiA 303 two LED can be controlled by the CANopen Stack. According to the current NMT and error state the LED can be switched on or off by a registered indication function (see coEventRegister_LED_RED() and coEventRegister_LED_RED()).

8.14 LSS Slave

For the LSS service contains an own LSS state machine, which is not connected to the NMT state machine.

Status	Definition
LSS Waiting	Normal operation
LSS Configuration	Configuration state, node-id and bitrate can be configured

The LSS master can switch the slave between this to states. The application can get informed in the callback which can be registered by coEventRegister_LSS().

The LSS slave has internally 3 Node-Id values:

Persistant Node-Id	Power-On Value, gets provides by the application
Pending Node-Id	Temporally Node-Id
Active Node-Id	Active Node-Id of the device

NMT state change and/or internal events can cause a copy procedure of the Node-Ids:

NMT Status	Persistant Node-Id	Pending Node-Id	Active Node-Id
Reset Application			
Reset Communication			
LSS Set Node-Id		Set new value	
LSS Store Node-Id			

The Active Node-Id is copy in Reset Communication from the Pending Node-Id. The switch state command Reset Communication has to be send by the NMT-Master.

If the device starts with Persistant Node-Id = 255 and get a valid node id by „LSS Set Node Id", an automatic state switch to Reset Communication is triggered by the state switch LSS State *Waiting*.

The Persistant Node Id has to be applied as Standard Node-Id by the application. If the Persistant Node-Id is saved in non volatile memory and changeable at run time, the application has to provide a function to provide the Standard Node-Id. Otherwise an incorrect Node-Id gets applied in Reset Application.

LSS Master commands g get indicated by a callback, which can be registered by *coEventRegister_LSS()*. If the LSS master sends a „LSS Store Command", the new Node-Id (=> Persistant Node-Id) has to be saved in non volatile memory, and provided by the function for the Node Id. If the Persistant Node-Id is supposed to constant, the "LSS Store Command" has to be aborted with an error code.

8.15 Configuration Manager

The Configuration manager module can be used in CANopen master applications only. It is used to configure NMT slaves by using appropriated DCF files. These DCF files can be present as ASCII files or as so-called Concise-DCF format files.

To be transferred to the NMT slaves the DCF has to be available as Concise-DCF in NMT masters object 0x1F22. If not available as Concise-DCF the function *co_cfgConvToConsive()* can be used to convert ASCII DCF files into Concise-DCF. Appropriate buffers have to be handed over in order to convert data partially.

Configuration is done for each single NMT slave by calling function *co_cfgStart()*. If objects 0x1F26 and 0x1F27 (expected configuration date/time)are available the function also does check the slave object 0x1020. If the object is not available or the slave configuration is not up to date, the configuration transfer does take place. The end of the transfer is signaled to the caller by the registered indication coEventRegister_CFG_MANAGER(). The indication will inform if the transfer was successful or not.

The configuration itself is using SDO transfer. For every to be configured node the according SDO client has to be configured on the NMT master. For example to configure node 32 the SDO Client 32 has to be available. Configuration of more than one NMT slave in parallel is possible.

Attention: While configuration transfer is in progress the SDO can not be used for other SDO transfers by the application.

8.16 Flying Master

To use the Flying Master functionality the object 0x1f80 must be present and the Flying Master bit has to be set. At startup of the network the device starts as a slave and starts the Flying Master Negotiation automatically. The result will be signaled by the callback function registered with coRegister_FLYMA(). If the node runs as a slave due to its priority, the application has to configure heartbeat monitoring for the active master. If the active master is lost, a new Flying Master Negotiation is started automatically.

8.17 Communication state

Changes in the communication state can be triggered by hardware events (Bus-Off, Error Passive, overflow, CAN message received, transmission interrupt) or by a timer (return from bus-off). These state changes are signaled by a registered indication function (see coEventRegister_COMM_EVENT()).

The following table describes which events cause a change of the communication state:

Event/Change of state	New state	Description
Bus-OFF	Bus-OFF	CAN controller is Bus-OFF, no communication possible
Bus-OFF Recovery	Bus-OFF	CAN controller tries to switch from bus-Off to active state
Return from Bus-OFF	Bus-On	CAN controller is ready to communicate and was able to receive or transmit at least 1 message
Error Passive	Bus-on, CAN passive	CAN controller is in error passive state
Error Active	Bus on	CAN controller is in error active state
CAN Controller overrun	-	Messages are lost in the CAN controller. The event is signaled at each loss of a message
REC-Queue full	-	Receive queue is full
REC-Queue overflow	-	Messages are lost because the receive queue is full. This event is signaled at each loss of a message.
TR-Queue full	Bus-Off/On, Tr-Queue full	Transmit Queue is full, the current message is saved, following message will not be saved
TR-Queue overflow	Bus-Off/On, Tr-Queue overflow	Transmit Queue is full, the message was not saved
TR-Queue empty	Bus-On, Tr-Queue ready	Transmit is ready to store messages (at least 50% free)

8.18 Sleep Mode for CiA 454 or CiA 447

Sleep Mode according to CiA-454 can be used as NMT slave or NMT master. The current Sleep mode phase can be evaluated or set by a user function registered with coEventRegister_SLEEP().

Sleep mode is commanded by the NMT master and consists of different stages:

NMT Master function	Stage/phase	Slave
coSleepModeCheck()	Sleep Check	Check if the Sleep mode can be entered by the slave. If not this is signaled to the NMT master
coSleepModeStart()	Sleep Prepare	Prepare the Sleep mode, bring down the application, but communication is still possible, start sleep timer 1
(timer controlled)	Sleep Silent	Transmitting over CAN is not anymore possible, but commands still can be received
(timer controlled)	Sleep	Sleep mode

After the master has initiated the "Sleep Prepare" phase the next stages are forced by a timer. All phases are the same for the NMT master and slave. The change into the next phase is signaled by the registered function. The application does not leave the indication function when it is in Sleep mode.
The application wakes up as soon as traffic on the CAN bus is recognized. Task of the application is it to keep all application data in the same state as just before the Sleep state. Then it calls *coSleepAwake()* once which leads to a Reset communication state on the node.
The function *coSleepModeActive()* can be used to check if one of the Sleep stages is active.

8.19 Startup Manager

To use the startup manager the following preconditions have to be met:

- Object 0x1f80 (NMT Master) must exist and it must be configured in the right way
- For each slave the properties have to be set in object 0x1f81 (Slave Assignment). The sub-index corresponds to the node-ID of the slave
- The boot time (object 0x1f89) must be set the largest possible boot time
- A Client SDO has to be provided for each Slave

The function *coManagerStart()* starts the boot-up process according to CiA 302-2. All required information are taken from the objects 0x1f80.. 0x1f89. Events like start, stop, error, or application interaction are signaled by the indication function that can be registered using coEventRegister_MANAGER_BOOTUP(). It is the task of the application to check and to update the slave firmware and to update the configuration. After the application has finished its tasks it may continue the boot-up process using the following functions:

Event	Task of application	Continuation with
CO_MANAGER_EVENT_UPDATE_SW	Check and update of slave Firmware	coManagerContinueSwUpdate
CO_MANAGER_EVENT_UPDATE_CONFIG	Update of slave configuration	coManagerContinueConfigUpdate
CO_MANAGER_EVENT_RDY_OPERATIONAL	Start node (transition to OPERATIONAL)	coManagerContinueOperational

9 Timer Handling

The Timer handling is based on a cyclic timer. Its interval can be individually defined for each application and the use of external timer is possible as well. A timer interval is called timer-tick and the timer-tick is the base for all timed actions in the CANopen Stack.

A new Timer is started by *coTimerStart()* and sorted into the linked timer list, so that all timed actions are sorted in this list. Thus after one timer-tick only the first timer has to be checked as the following timers cannot be expired yet.

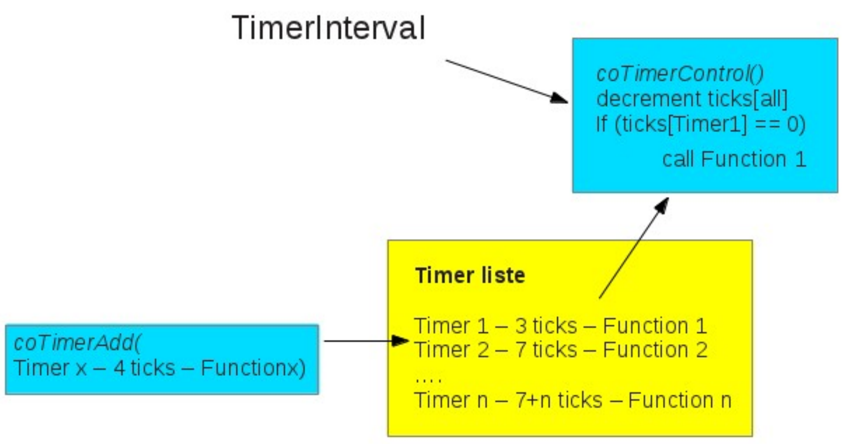


Illustration 12: Timer Handling

The timer structure must be provided by the calling function. This means also that there is no limitation of the number of timers.

It might happen that not all times will be a multiple of a timer tick. In this case it is possible to specify if the timer time shall be rounded up or down. This is done when starting the timer by using the function *coTimerStart().*

10 Driver

The driver consists of a part for the CPU and a part for the CAN controller.

CPU driver

The task of the CPU driver is to provide a constant timer tick. It can be created by a hardware interrupt or derived from another application timer.

CAN driver

The task of the CAN driver is to handle and to configure the CAN controller, to send and to receive CAN messages and to provide the current state of the CAN. The buffer handling is done by the CANopen Protocol Stack.

10.1 CAN Transmit

Messages to be transmitted are transferred by the CANopen stack into the transmit queue. Transmission itself is then started by the function *codrvCanStartTransmission()*. Transmission of all messages is interrupt driven. The function *codrvCanStartTransmission()* has only to issue or simulate an TX interrupt.

The TX interrupt service function has to use *codrvCanTransmit()* to get the next message from the queue, program the CAN controller and transmit it. This is done until the TX queue is empty.

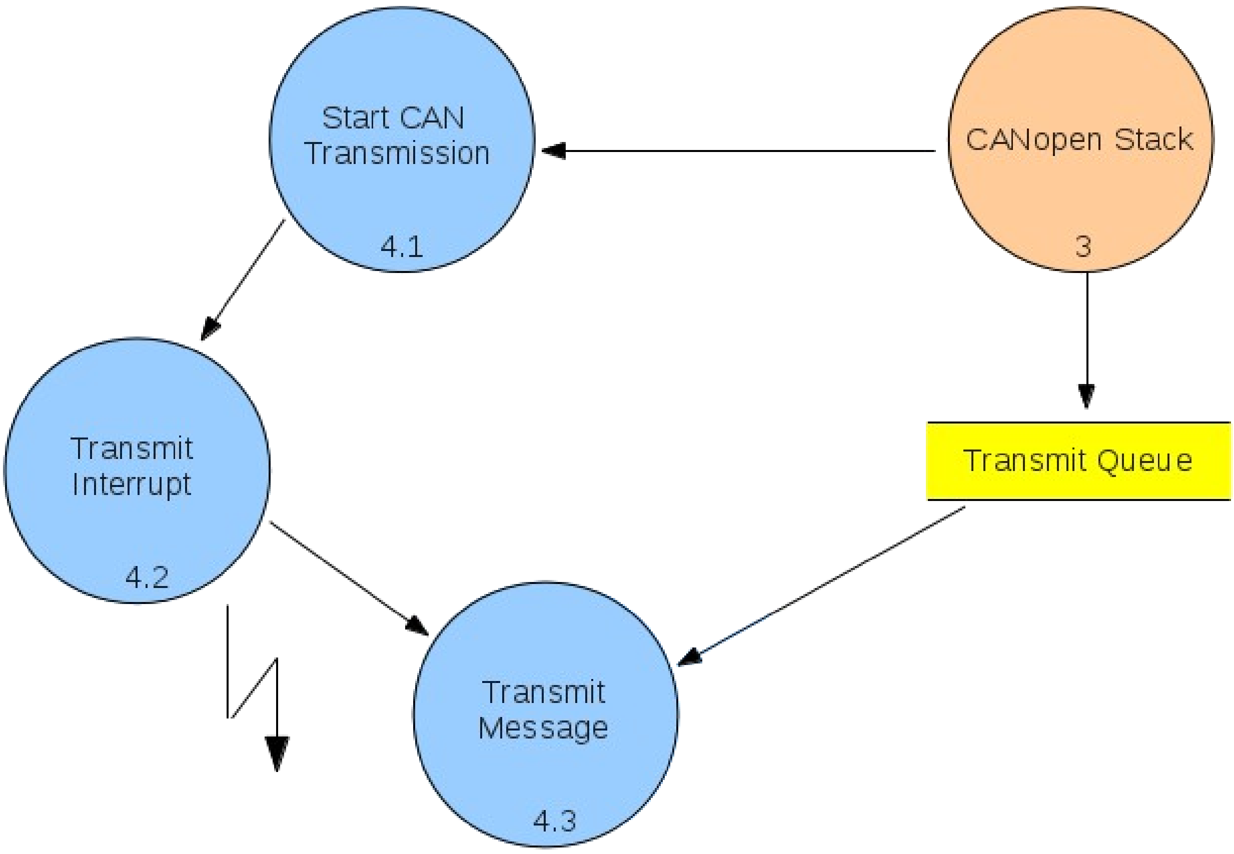


Illustration 13: CAN Transmit

10.2 CAN Receive

Reception of CAN messages is interrupt driven. The received CAN message is transferred into the RX queue and can be later used by the CANopen stack.

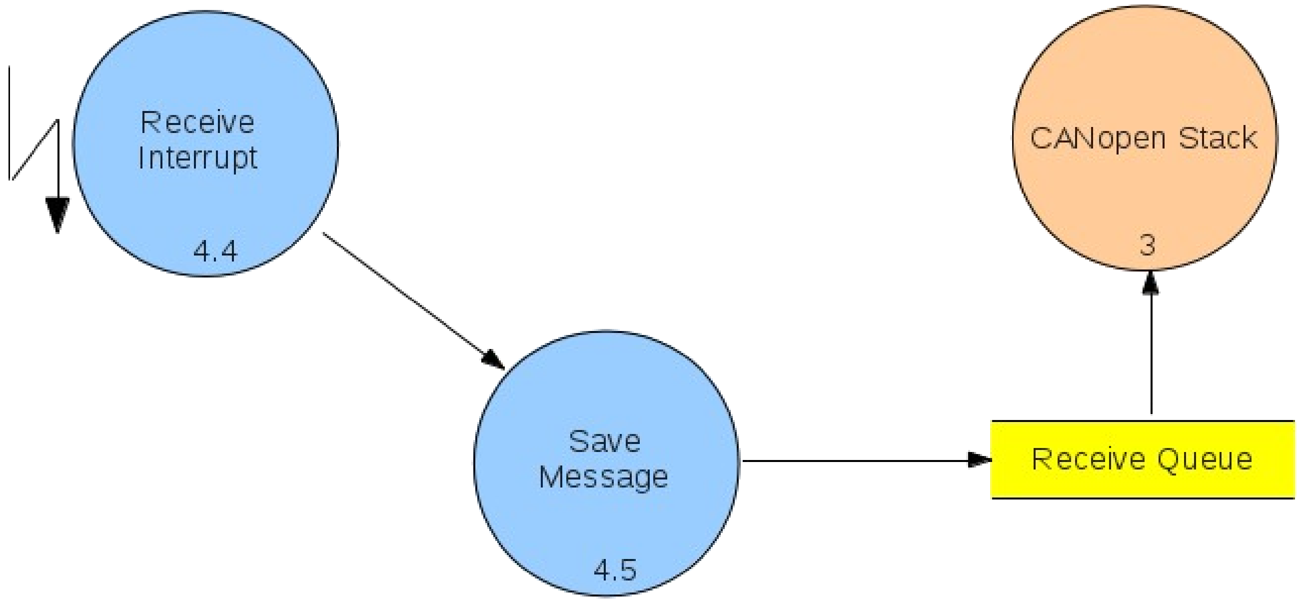


Illustration 14: CAN Receive

11 Using operation systems

To use the CANopen stack together with a real time operational system (RTOS) there are two possibilities:

1. Use of the CANopen Stack within one task only and cyclic call of the central stack function
2. Separation into multiple tasks

   This requires an inter task communication.

11.1 Separation into multiple tasks

If the CANopen Stack is called from multiple tasks, polling of the central stack function is no longer necessary, but it is this the central function which has to be called at the following events:

- CAN Transmission interrupt
- CAN Receive interrupt
- CAN State interrupt (if supported)
- Timer interrupt (or timer tick signal, timer task)

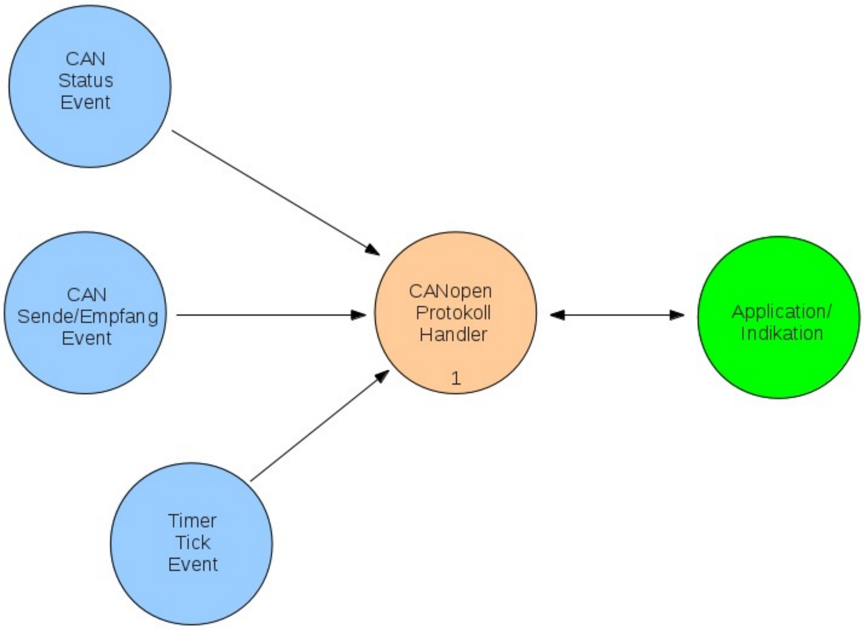


Illustration 15: Process Signal Handling

The implementation of the inter task communication and handling depends of the used operation system.

Macro	Usage	Meaning
CO_OS_SIGNAL_WAIT()	coCommTask()	Waiting for any signal
CO_OS_SIGNAL_TIMER()	Timer handler	Timer Tick
CO_OS_SIGNAL_CAN_STATE()	CAN status interrupt	Changed CAN Status
CO_OS_SIGNAL_CAN_RECEIVE()	CAN Receive Interrupt	New CAN message received
CO_OS_SIGNAL_CAN_TRANSMIT()	CAN Transmit Interrupt	New CAN message transmitted

11.2 Object dictionary access

If the access to the CANopen stack is split into multiple tasks, the access to the object dictionary has to be protected to prevent simultaneous accesses form different tasks. The following macros are available for that:

CO_OS_LOCK_OD	Lock of the object dictionary
CO_OS_UNLOCK_OD	Unlock of the object dictionary

These macros have to be implemented depending on the operating system and have to be called from application as well when a value of the object dictionary is accessed.

Within the stacks the lock resp. unlock is done immediately before and after the access to the objects.

11.3 Mailbox-API

The mailbox-API offers an alternative API for the functions and indications of the CANopen stack. Using this approach the CANopen stack runs in a separate thread/task[5]. Any arbitrary number of application threads can be created that can send commands to the CANopen thread via a message queue.

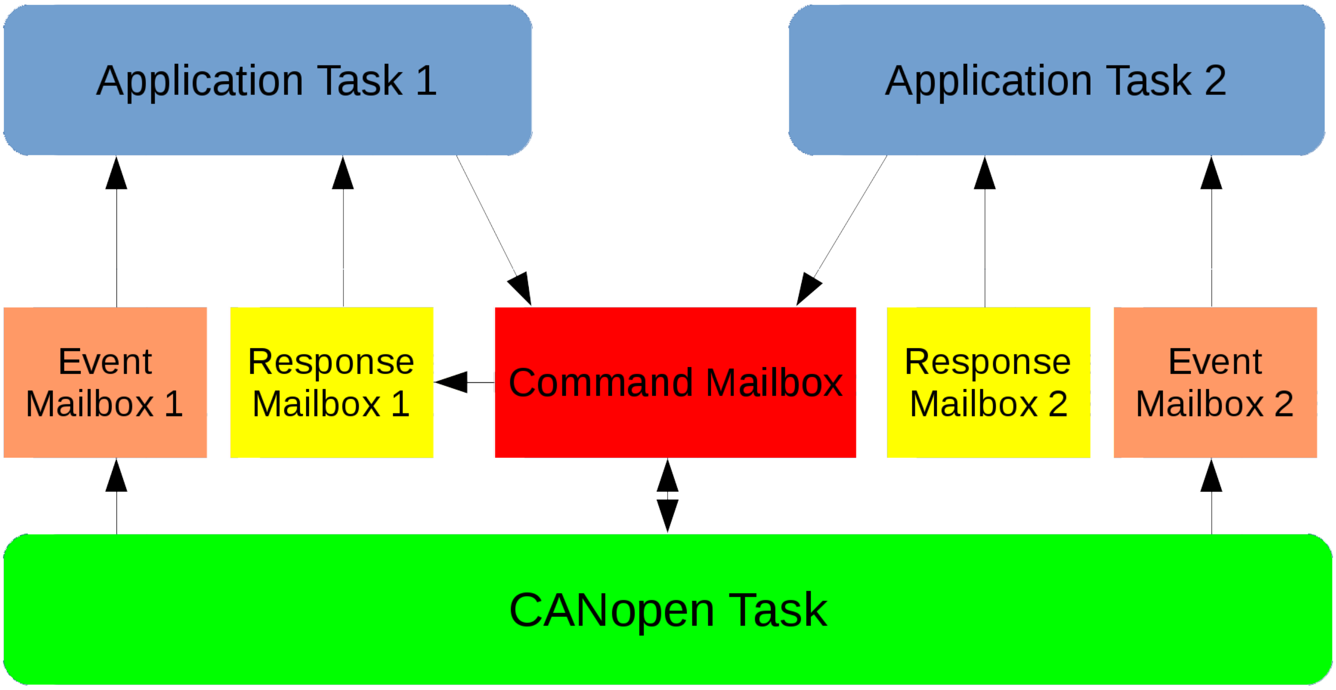


Illustration 16: Mailbox-API

The CANopen thread sends a response for each command back to the application thread via a response queue which contains the return value of the function. Additionally, indications for various events can be received via an event queue. It is possible to configure which events are sent to each application thread. The event handling replaces the indication functions of the normal function API.

Currently the Mailbox-API is ported to the operating systems QNX, Linux and RTX64 but any operating system that provides queues can be supported.

5 The term thread is used for further explanations. It depends on the operating system if a thread or task is used.

11.3.1 Creation of an application thread

Each application thread consists of an initialization and a cyclic main part. In the initialization part the thread has to connect to the command queue of the CANopen thread and optionally thread-specific response and event queues can be created as shown in the following example:

```
/* connect to command mailbox */
mqCmd = Mbx_Init_CmdMailBox(0);
if (mqCmd < 0)  {
      printf("error Mbx_Init_CmdMailBox() - abort\n");
      return(NULL);
}

/* create response mailbox */
mqResp = Mbx_Init_ResponseMailBox(mqCmd, "/respMailbox1");
if (mqResp < 0)  {
      printf("error Mbx_Init_ResponseMailBox() - abort\n");
      return(NULL);
}

/* create response mailbox */
mqEvent = Mbx_Init_EventMailBox(mqCmd, "/eventMailbox1");
if (mqEvent < 0)  {
      printf("error Mbx_Init_EventMailBox() - abort\n");
      return(NULL);
}
```

After creating the mailboxes the event mailbox has to be configured in order to defined which events shall be sent to the application:

```
/* register for Heartbeat events like Bootup, HB started or HB lost */
ret = Mbx_Init_CANopen_Event(mqCmd, mqEvent, MBX_CANOPEN_EVENT_HB);
if (ret != 0)  { printf("error %d\n", ret); };

/* register for received PDOs */
ret = Mbx_Init_CANopen_Event(mqCmd, mqEvent, MBX_CANOPEN_EVENT_PDO);
if (ret != 0)  { printf("error %d\n", ret); };
```

11.3.2 Sending commands

For all basic CANopen functions and important CANopen master functions mailbox commands are available. To send such a command the corresponding struct have to be filled with the arguments of the corresponding CANopen function as shown in the following example:

```
/*--------------------------------------------------------------*
 * Send an emergency message
 * corresponds to: coEmcyWriteReq(errorCode, pAdditionalData);
 *---------------------------------------------------------------------*/
MBX_COMMAND_T   emcy;
emcy.data.emcyReq.errCode = 0xff00;
memcpy(&emcy.data.emcyReq.addErrCode[0], "12345", 5);
ret = requestCommand(mqResp, MBX_CMD_EMCY_REQ, &emcy);


/*---------------------------------------------------------------------*
 * Send a NMT request to start all nodes including the master
 * corresponds to: coNmtStateReq(node, state, masterFlag);
 *---------------------------------------------------------------------*/
MBX_COMMAND_T   nmt;
nmt.data.nmtReq.newState = CO_NMT_STATE_OPERATIONAL;
nmt.data.nmtReq.node = 0;
nmt.data.nmtReq.master = CO_TRUE;
ret = requestCommand(mqResp, MBX_CMD_NMT_REQ, &nmt);
```

The return value of requestCommand() is a number which is automatically incremented. This number is also sent back by the Response from the CANopen thread. This allows to keep track of commands and their returns value (of the underlying CANopen functions).

The following CANopen functions are currently supported by the Mailbox-API:

CANopen function	Command
coEmcyWriteReq()	MBX_CMD_EMCY_REQ
coPdoReqNr()	MBX_CMD_PDO_REQ
coNmtStateReq()	MBX_CMD_NMT_REQ
coSdoRead()	MBX_CMD_SDO_RD_REQ
coSdoWrite()	MBX_CMD_SDO_WR_REQ
coOdSetCobId()	MBX_CMD_SET_COBID
coOdGetObj_xx()	MBX_CMD_GET_OBJ
coOdPutObj_xx()	MBX_CMD_PUT_OBJ
7 coLss... Functions	MBX_CMD_LSS_MASTER_REQ

Please refer to the reference manual for an explanations of the functions(commands) and the return values(responses).

11.3.3 Reception of events

If events are registered by an application thread the can be received using Mbx_WaitForEventMbx(). All events correspond to the indication functions of the function API and the members of the event structure correspond to the arguments of the indication functions.

```
/* wait for new events for 0ms*/
if (Mbx_WaitForEventMbx(mqEvent, &event, 0) > 0)  {
        printf("event %d received\n", event.type);

        /* message depends on event type */
        switch (event.type)  {
        /* Heartbeat Event like Bootup, heartbeat started or Heartbeat lost */
        case MBX_CANOPEN_EVENT_HB:
            printf("HB Event %d node %d nmtState: %d\n",
            response->event.hb.state,
            response->event.hb.nodeId,
            response->event.hb.nmtState);
            break;

        /* PDO reception */
        case MBX_CANOPEN_EVENT_PDO:
            printf("PDO %d received\n", response->event.pdo.pdoNr);
            break;

        /* see example for more events */
        default:
            break;
        }
}
```

12 Multi-Line Handling

The usage of the Multi-Line stack is the same as with the single-line version. All described can be used with multiple CAN lines. All data of all lines are handled separately so that all lines can be run independent of each other. The object dictionary for multi-line applications is created in a single project of the CANopen DeviceDesigner but each line is handled in a separate way.

Each API functions has an additional argument in the beginning which indicates the like as an UNSIGNED8 value starting at 0. The applies for all stack functions and all indication functions.

Examples for multi-line applications can be found in example_ml/*xxx*.

13 Multi-Level Networking – Gateway Functionality

The object 0x1F2C is required to use the gateway functionality. This object defines routes that specific which network can be reach at which CAN interface.

13.1 SDO Networking

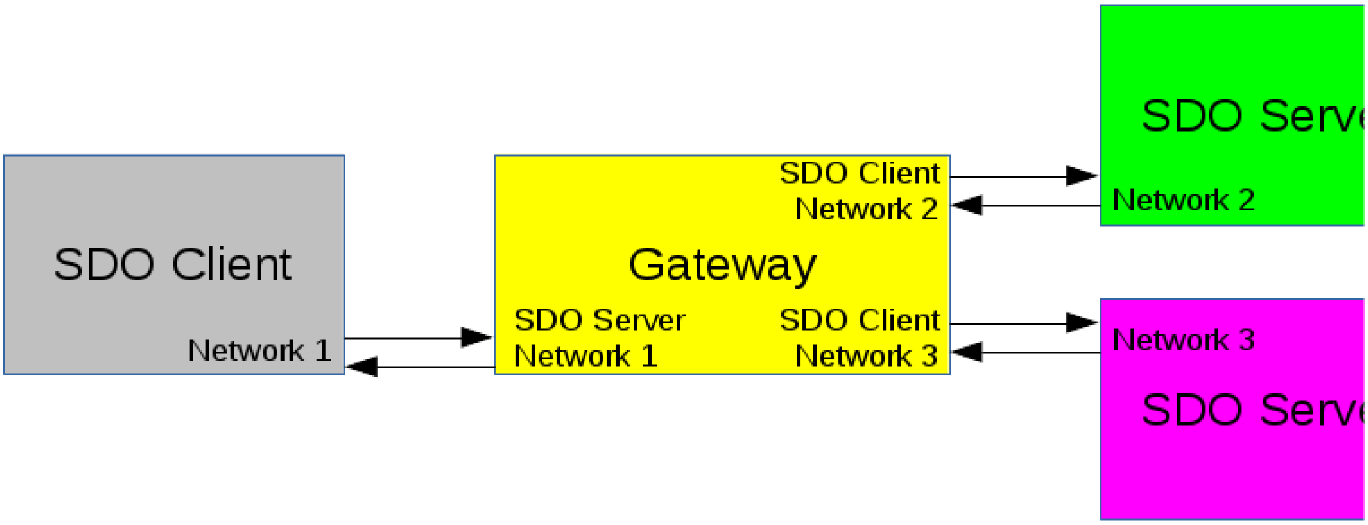


Illustration 17: Multi-Level Networking

The SDO client initiates a connection to the gateway. This initiation message contains the target network ID and the target node-ID.

All following SDO requests to the gateway are now forwarded to the target node. The gateway receives the SDO request as an SDO server and uses an SDO client connection in the remote network to reach the other node.

The CANopen stack needs to know which SDO client it shall use. It can specified for each connection. The application programmer may register an indication function by register_GW_CLIENT() and in callback function the application may specify an SDO client. If no function is registered always SDO client 1 is used. If SDO client 1 busy, no connection may be established. The COB-IDs for the SDO client are configured automatically but not reset at the end of a transfer.

13.2 EMCY Networking

Object 0x1f2f is required to support Emergency routing. It contains a bit-coded information about the networks the EMCY message shall be forwarded to. The sub indices correlate to the Emergency Consumer list (object 0x1028) and are evaluated in parallel.

13.3 PDO Forwarding

PDO Forwarding is handled automatically for all objects in the rage 0xB000 to 0xBFFF, which are mapped into a Transmit or receive PDO. In the CANopen DeviceDesigner you have to these objects only in one line, and declare them as “shared in all lines”. In general one Receive PDO can only mapped to exactly one Transmit PDO, because the forwarding list is deposited with the Receive PDO.

For static PDOs this list can not be modified at run-time, even if the mapping of the Transmit PDO has being changed. The update of the forwarding takes place after every modification of the PDO mapping

14 Example implementation

The CANopen stack comes with multiple example for a fast implementation of a CANopen device.

The necessary steps depend on the development environment, but the steps in general are identical. It is shown using the example slave1. It can be copied or used directly.

1. go to folder example_sl/slave1
2. configuration of CANopen services and object dictionary

   - start *CANopen DeviceDesigner*

   - File->OpenProject – open project file slave1.cddp

   - Tab General Settings - define number of send and receive buffers and the number of used indication functions

   - Tab Object Dictionary – optionally add objects and services

   - Tab Device Description – add entries for the EDS files

   - File->Generate Files – generate object dictionary and configuration files (.c/.h)

   - File->Save Project – save project
3. Add CANopen source files to project (in IDE) or makefile

   - Files in colib_sl/src (CANopen Stack)

   - Files in colib_sl/inc (CANopen Stack public Header)

   - Files in example_sl/slave1 (example application)

   - Files in codrv_sl/<drivername> (driver)
4. Set include paths

   - example_sl/slave1

   - colib_sl/inc

   - codrv_sl/<drivername>
5. build (compile and link) the project

Now a ready-to-run CANopen project is available, that can be modified according to the requirements of the application.

For own implementations, please ensure, that you include gen_define.h always before co_canopen.h in your own sources.

Files in example project slave1

gen_define.h	generated file by CANopen DeviceDesigner, contains configuration for CANopen stack
gen_objdict.c	generated files by CANopen DeviceDesigner, contains object dictionary and initialization functions.
main.c	Main part of the program
Makefile	Makefile
slave1.cddp	Project file for *CANopen DeviceDesigner*
slave1.eds	EDS File, generated by *CANopen DeviceDesigner*

14.1 Hardware Adaptation and Development Environment

All hardware based files are located in the codrv_sl/<driver_name> directory. This includes the files for the CAN controller handling as well as the files for the hardware definitions and configuration, depending on the development environment.

The files for the hardware definition and configuration are typically named cpu_<processor>.c and cpu_<processor>.h. These two files have to be adapted for the own hardware requirements.

We suggest to copy these files under a new name into your project and use them instead of the original files to prevent overwriting them with a stack update.

15 C#-Wrapper

For Windows, as for Mono under Linux, there is a C#-Wrapper available. The CANopen C stack is available thru a dynamic linked library (DLL). The C# sharp wrapper uses this DLL to access the provided CANopen functions.

All C#Wrapper methods are static and implemented in one class. The class methods use the same names as the ANSI-C functions.

Examples:

```
CANopen.coEventRegister_NMT()==coEventRegister_NMT()
CANopen.coEmcyWriteReq()== coEmcyWriteReq()
CANopen.coCommTask()== coCommTask()
…
```

All return values and parameters are equivalent to the C version, so the user manual and the reference manual of C implementation can be used.

16 Step by Step Guide – using CANopen Services

16.1 SDO server usage

Configuration using the CANopen DeviceDesigner

- For each SDO server create an SDO object in the range of 0x1200 to 0x127F
  (hint: No need to set COB-IDs, this is done by the application program)
- If SDO Block transfer should be used, set parameters:
  - block size to be used
  - usage of CRC yes/no

Configuration of the application:

- register functions for read, write or test using coEventRegister_SDO_SERVER_READ() / coEventRegister_SDO_SERVER_WRITE() / coEventRegister_SDO_SERVER_CHECK_WRITE())
- COB-Id for SDO number one is set automatically according to the CANopen node-Id
- set COB-Id for all other server SDOs or alternatively wait until they are configured at run time by the NMT master

Usage in the application:

- Asynchronous via the registered indication function. The return value of the indication function affects the responses of the SDO transfer.

16.2 SDO client usage

Configuration using the CANopen DeviceDesigner

- For each SDO client create an SDO object in the range of 0x1280 to 0x12FF
  (hint: no need to set COB-IDs, this has to be done by the application program)
- If SDO block transfer should be used, set parameters:
  - block size to be used for the transfer
  - number of bytes when block transfer should be used. If smaller normal segmented is used
  - usage of CRC yes/no

Configuration of the application:

- Register indication functions for the result of a read or write request (coEventRegister_SDO_CLIENT_READ() / coEventRegister_SDO_CLIENT_WRITE())
- set COB-Ids for all client SDOs or set these just before the request is used

Usage in the application:

- set COB-Ids appropriate for the server to be requested
- start the request using coSdoRead(), coSdoWrite(), coSdoDomainWrite()
- get the result via the registered indication function
- usage of domain transfers (coSdoDomainWrite()) can add an additional indication function, it is called after defined number of messages was transmitted, e.g. to reload the domain buffer

16.3 USDO Server Utilization

Setup in the CANopen DeviceDesigner:

– *define the number of indication functions*

Setup in the application:

– Setup indication functions for read/write/write-test (coEventRegister_USDO_SERVER_READ() / coEventRegister_USDO_SERVER_WRITE() / coEventRegister_USDO_SERVER_CHECK_WRITE())

Usage in the application:

– takes place asynchronous at USDO reception
the return value has effect on the USDO Transfer Response

16.4 USDO Client Utilization

Setup in the CANopen DeviceDesigner:

– *define the number of the indication functions*

Setup in the application:

– Setup indication functions for the result of read/write (coEventRegister_USDO_CLIENT_READ() / coEventRegister_USDO_CLIENT_WRITE())

Usage in the application:

– Start transfer (coUsdoRead(), coUsdoWrite(), coUsdoDomanWrite())

– The result will be delivered by the setup indication function

– When using the domain transfer (coUsdoDomanWrite()) an additionally indication can be defined, which will be called after a defined number of Can telegrams for example to update the transmit buffer.

16.5 Heartbeat Consumer

Configuration using the CANopen DeviceDesigner:

- for each heartbeat consumer create a sub-index entry in object 0x1016 using the CANopen DeviceDesigner
- Node number and consumer time can be configured directly in this sub-index entry

Configuration of the application:

- Register the indication function for heartbeat events
- eventually set consumer time and node-Id again

Usage in the application:

- Consumer time monitoring starts when the first heartbeat of the supervised node arrives
- Each heartbeat event, started, offbeat, changed node state, is signaled via the registered indication function

16.6 Emergency Producer

16.6.1 CANopen classic

Configuration using the CANopen DeviceDesigner:

- Create the Emergency Producer object 0x1014 using the CANopen DeviceDesigner
- Create the Error History object 0x1003 with n sub-indices according the application requirements using the CANopen DeviceDesigner

Configuration of the application:

- Register the indication function using coEventRegister_EMCY() which is used to get the manufacturer specific data

Usage in the application:

- Sending the EMCY by calling *coEmcyWriteReq()*
- The registered indication function is called at PDO errors (to much, to less data), CAN or heartbeat errors

16.6.2 CANopen FD

Setup in the CANopen DeviceDesigner:

– Setup emergency producer object (0x1014)
– Setup error history objects

Setup in the application:

– Setup indication function for set manufacturer specific data (coEventRegister_EMCY())

Usage in the application:

– Call via coEmcyWriteReq()
– registered indication will be called when PDO errors (to much/less data) or CAN/Heartbeat errors occur
–

16.7 Emergency Consumer

Configuration using the CANopen DeviceDesigner:

- Create the Emergency Consumer object 0x1028 using the CANopen DeviceDesigner
- Fill in the Emergency Consumer COB-IDs. Sub-index corresponds to the external node-Id

Configuration of the application:

- Register the indication function using coEventRegister_EMCY_CONSUMER() which is called when an EMCY arrives

Usage in the application:

- Registered indication function is called if a configured EMCY consumer entry matches a received EMCY

16.8 SYNC Producer/Consumer

Configuration using the CANopen DeviceDesigner:

- Create the SYNC object 0x1005 using the CANopen DeviceDesigner
- define if it is used as Consumer or Producer (defined by bit 30)
- For SYNC Producer configure producer time at object 0x1006 in µseconds

Configuration of the application:

- Register the indication function using coEventRegister_SYNC() for received SYNC messages
- Register the indication function using coEventRegister_SYNC_FINISHED() for actions to be done after the SYNC handling the stack has already done

Usage in the application:

- Registered functions are called after a SYNC message was received

16.9 PDOs

16.9.1 Receive PDOs

Configuration using the CANopen DeviceDesigner:

- Create objects which should be received by PDO within the manufacturer (0x2000 to 0x5FFF) or profile (0x6000) area
- Set the PDO mapping flag of these objects to allowed, RPDO or TPDO
- Create PDO communication parameters for each PDO (objects 0x1400 to 0x15FF):
    - Set transmission type – for synchronous PDOs the SYNC object must be created as well (see 16.8)
    - Set event timer value in milliseconds
- Configure the PDO mapping for this PDO (objects 0x1600 to 0x17FF)
- Select mapping type static or dynamic using the tab “Mask”

Configuration of the application:

- Configure or modify used COB-Id. RPDO1 to RPDO4 are configured according the “predefined connection set” if not changed
- Register indication function for received asynchronous PDOs using coEventRegister_PDO()
- If required register indication function for the event timer used for monitoring the reception of the RPDO using coEventRegister_PDO_REC_EVENT()
- If required register the EMCY in case wrongly configured PDOs should be reported by sending the appropriate EMCY code
- If required register the SYNC receive indication using coEventRegister_PDO_SYNC()

Usage in the application:

- Registered indication functions are automatically called if a configured RPDO is received. Object dictionary entries are already updated

16.9.2 Transmit PDOs

Configuration using the CANopen DeviceDesigner:

- Create objects which should be transmitted by PDO within the manufacturer (0x2000 to 0x5FFF) or profile (0x6000) area
- Set the PDO mapping flag of these objects to allowed, RPDO or TPDO
- Create PDO communication parameters for each PDO (objects 0x1800 to 0x19FF):
    - Set transmission type – for synchronous PDOs the SYNC object must be created as well (see 16.8)
    - Set event timer value in milliseconds
    - Set Inhibit Time in 100µseconds
    - Set SYNC Start value if used
- Configure the PDO mapping for this PDO (objects 0x1a00 to 0x1bFF)
- Select mapping type static or dynamic using the tab “Mask”

Configuration of the application:

- Configure or modify used COB-Id. TPDO1 to TPDO4 are configured according the “predefined connection set” if not changed

Usage in the application:

- Transmit a PDO
    - update the object dictionary data (which are mapped into a TPDO)
    - PDOs with transmission type 0 – acyclic, 254 and 255 – asynchronous are sent by calling *coPdoWriteNr()* or *coPdoWriteIndex()*
    - Synchronous PDOs with transmission type 1 to 240 are sent automatically when SYNC arrives

16.10 Dynamic objects

Activation in CANopen DeviceDesigner:

- Optional Services → Use Dynamic Objects

Configuration of the application:

- Initialization of dynamic object dictionary *coDynOdInit()*
- Add an object to object dictionary *coDynOdAddIndex()*
- Add a sub object to object dictionary *coDynOdAddSubIndex()*

Usage in the application:

- dynamic objects can be accessed in the application in the same way as static created objects by the CANopen DeviceDesigner

16.11 Object Indication

Configuration using the CANopen DeviceDesigner:

- Configure the maximum number of objects used this way
  example: #define CO_EVENT_OBJECT_CHANGED 5

Configuration of the application:

- Register an indication function using coEventRegister_OBJECT_CHANGED()

Usage in the application:

- The registered function is called if the object was changed by SDO write access or by a received PDO

16.12 Configuration Manager

Configuration using the :

- Create objects 0x1F22 and 0x1F23 (Conceive DCF) with corresponding sub indices
- Create objects 0x1F26 and 0x1F27 (configuration date/time) optionally
- Create all required SDO Client(s) 0x1280..0x12ff

Configuration of the application:

- register an indication function using registerEvent_CFG_MANAGER()
- Put Concise DCF files into the object 0x1F22
- or read DCF file and convert them do concise DCF using *co_cfgConvToConsive()*
- and put concise data to 0x1F22

Usage in the application:

- Start the configuration for each node by *co_cfgStart().*
- A completion is signaled by the registered indication function.

17 Directory structure

codrv_sl/xxx	Hardware specific CANopen Single-line driver
codrv_sl/common	Common CANopen Single-line driver files
colib_sl/inc	CANopen Single-line protocol Stack Header
colib_sl/src	CANopen Single-line protocol Stack sources and internal headers
colib_sl/profile	CANopen Single-line Profile
colib_sl/csharp_wrapper	CANopen Single-line C# Wrapper
example_sl	CANopen Single-line examples
codrv_ml/xxx	Hardware specific CANopen Multi-line driver
codrv_ml/common	Common CANopen Multi-line driver files
colib_ml/inc	CANopen Multi-line protocol Stack Header
colib_ml/src	CANopen Multi-line protocol Stack sources and internal headers
colib_ml/profile	CANopen Multi-line Profile
example_ml	CANopen Multi-line examples
cofddrv_sl/xxx	Hardware specific CANopenFD Single-line driver
cofddrv_sl/common	Common CANopenFD Single-line driver files
cofdlib_sl/inc	CANopenFD Single-line Single-line protocol Stack Header
cofdlib_sl/src	CANopenFD Single-line protocol Stack sources and internal headers
examplefd_sl	CANopenFD Single-line examples
cofddrv_ml/xxx	Hardware specific CANopenFD Multi-line driver
cofddrv_ml/common	Common CANopenFD Multi-line driver files
cofdlib_ml/inc	CANopenFD Multi-line protocol Stack Header
cofdlib_ml/src	CANopenFD Multi-line protocol Stack sources and internal headers
examplefd_ml	CANopenFD Multi-line examples
ref_man	Reference Manual
user_man	User Manual

Appendix

SDO Abort codes

RET_TOGGLE_MISMATCH	0x05030000
RET_SDO_UNKNOWN_CCS	0x05040001
RET_SERVICE_BUSY	0x05040001
RET_OUT_OF_MEMORY	0x05040005
RET_SDO_TRANSFER_NOT_SUPPORTED	0x06010000
RET_NO_READ_PERM	0x06010001
RET_NO_WRITE_PERM	0x06010002
RET_IDX_NOT_FOUND	0x06020000
RET_OD_ACCESS_ERROR	0x06040047
RET_SDO_DATA_TYPE_NOT_MATCH	0x06070010
RET_SUBIDX_NOT_FOUND	0x06090011
RET_SDO_INVALID_VALUE	0x06090030
RET_MAP_ERROR	0x06040042
RET_PARAMETER_INCOMPATIBLE	0x06040043
RET_ERROR_PRESENT_DEVICE_STATE	0x08000022
RET_VALUE_NOT_AVAILABLE	0x08000024