# User Manual

# CANopen Master/Slave Protocol Stack

V 2.7.0

## 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<br>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 | |

## Table of Contents

## 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 Protocol Stack provides communication services for a CANopen compliant communication of devices and enables the fast and straight-forward integration of CANopen into devices. All services of CiA 301 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

- Separation of hardware-dependent and hardware-independent part with defined interface
- ANSI-C
- Compliance to mandatory MISRA-C rules
- support of all CiA 301 services
- configurable and scalable
- facility to add extension modules, e.g. for advanced master services
- compliant to CiA-301 V4.2
- flexible user interface
- static and dynamic object dictionary
- LED CiA-303

CANopen Services of the variants

| Service | Basic Slave | Master/Slave | Manager |
|---|---|---|---|
| SDO Server | 2 | 128 | 128 |
| SDO Client |  | 128 | 128 |
| SDO expedited/segmented/block | ●/●/- | ●/●/○ | ●/●/○ |
| PDO Producer | 6 | 512 | 512 |
| PDO Consumer | 6 | 512 | 512 |
| PDO Mapping | Static | Static/dynamic | Static/dynamic |
| MPDO Destination Mode |  | ○ | ○ |
| MPDO Source Mode |  | ○ | ○ |
| SYNC Producer |  | ● | ● |
| SYNC Consumer | ● | ● | ● |
| Time Producer |  | ● | ● |

| Service | Basic Slave | Master/Slave | Manager |
|---|---|---|---|
| Time Consumer | | ● | ● |
| Emergency Producer | ● | ● | ● |
| Emergency Consumer | | 127 | 127 |
| Guarding Master | | | ● |
| Guarding Slave | | ● | ● |
| Bootup Handling | | ● | ● |
| Heartbeat Producer | ● | ● | ● |
| Heartbeat Consumer | | 127 | 127 |
| NMT Master | | ● | ● |
| NMT Slave | ● | ● | ● |
| LED CiA-303 | ● | ● | ● |
| LSS CiA-305 | | ● | ● |
| Sleep Mode according CiA-454 | ● | ● | ● |
| Master Bootup CiA-302 | | | ● |
| Configuration Manager | | | ● |
| Flying Master | | ○ | ○ |
| Redundancy | | ○ | ○ |
| Safety (SRDO) | ○ | ○ | ○ |
| Multiline | | ○ | ○ |
| CiA-401 (U8/INT16) | ○ | ○ | ○ |
| CiA-xxx | ○ | ○ | ○ |

**● - included, ○ - optional**

# 3 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 within the stack
- each service provides its own initialization function

The

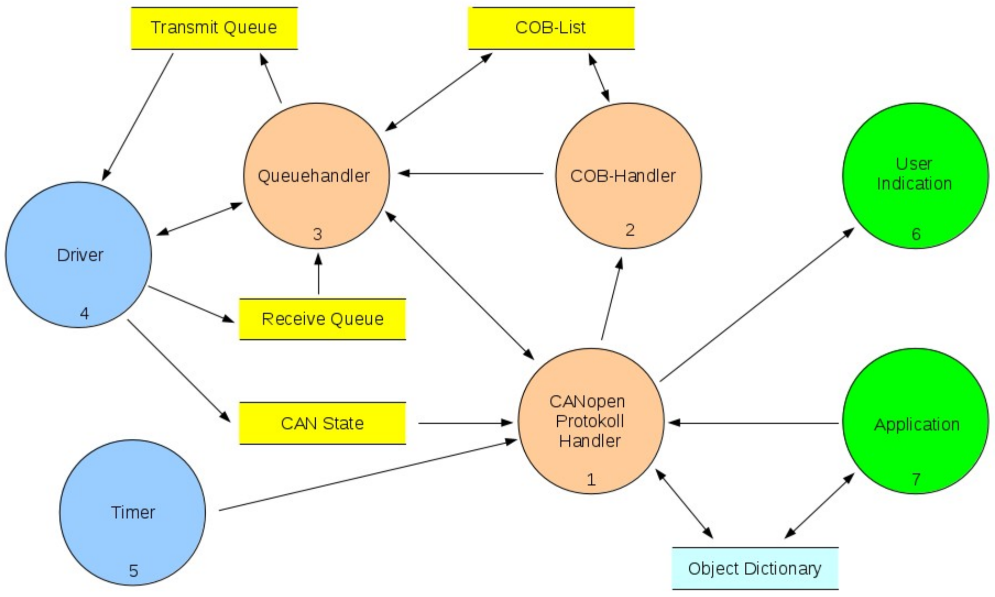


*Illustration 1: Module Overview*

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.

When 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).

# 4 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 Status changed | CAN Status | |
| EMCY | automatically generated Emergency message shall be sent | Error Code<br>Pointer to additional bytes | Send Emcy/Discard Emcy |
| EMCY_CONSUMER | Emergency Consumer message received | Node Id<br>Error Code<br>Error Register<br>Additional Bytes | |
| LED | Set red/green LED | On/off | |
| ERRCTRL | Heartbeat/Bootup State | Node Id<br>HB State<br>NMT Statue | |
| NMT | NMT State changed | new NMT state | Ok/not Ok |
| PDO | asynchronous PDO received | PDO Number | |
| PDO_SYNC | synchronous PDO received | PDO Number | |
| PDO_REC_EVENT | Time Out for PDO | PDO Number | |
| SDO_SRV_READ | SDO Server Read Transfer finished | SDO Server Number<br>index<br>subindex | Ok/SDO abort code/Split indication |
| SDO_SRV_WRITE | SDO Server Write Transfer finished | SDO Server Number<br>index<br>subindex | Ok/SDO abort code/Split indication |
| SDO_SRV_CHECK_WRITE | SDO Server Check Write Transfer | SDO Server Number<br>index<br>subindex<br>pointer to received data | Ok/SDO abort code |
| SDO_CLIENT_READ | SDO Client Read Transfer finished | SDO Client Number<br>index | |

| **EVENT_TYPE** | **Event** | **Parameters** | **Return value** |
|---|---|---|---|
| | | subindex<br>number of data<br>result | |
| SDO_CLIENT_WRITE | SDO Client Write Transfer finished | SDO Client Number<br>index<br>subindex<br>result | |
| OBJECT_CHANGED | Object was changed by SDO or PDO access | index<br>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 |

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 | Return value of the function shall be evaluated<br>Indication functionality may NOT be executed. |
| | CO_TRUE | Return value of the function is not evaluated.<br>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.

# 5 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

## 5.1 *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.

## 5.2 *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 writeable |
| 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 |

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

## 5.3 *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 |

## 5.4 *Strings and Domains*

Strings are handled in 2 different ways:

– Constant strings are put into the object dictionary. It exists a list of pointers to the strings and a list of size information. Both lists are constant and cannot be modified.

– Variable strings are handled like domains

The address and size of a domain can be configured at run-time using the function *coOdDomainAddrSet()* . All domains and their sizes are stored in domain lists.

### 5.4.1 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.

N.B.! 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.

## 5.5 Dynamic Object dictionary

### 5.5.1 Managed by Stack functions

Objects at the manufacturer and profile specific area can also be created dynamically at runtime. 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()[1]. 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 informations 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/dynod.

### 5.5.2 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 icoDynOdGetObjDescPtr(                /* get Object description */
        UNSIGNED16 index,                   /* index */
         UNSIGNED8 subIndex,                /* subindex */
         CO_CONST CO_OBJECT_DESC_T **pDescPtr
UNSIGNED8 icoDynOdGetObjAddr(               /* get address of object */
          CO_CONST CO_OBJECT_DESC_T     *pDesc /* pointer for description index */
UNSIGNED32 icoDynOdGetObjSize(              /* 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/dynod_appl may be used as a template for this functionality.**

It is not possible to mix both functionalities.

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

# 6 CANopen Protocol Stack Services

## 6.1 *Initialization functions*

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

| | |
|---|---|
| *coInitCanOpenStack()* | 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 |

### 6.1.1 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 registerEvent_NMT)) is called.

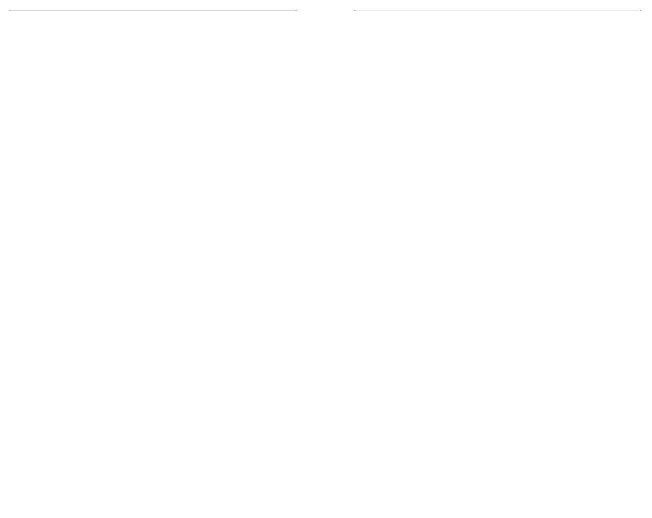


*Illustration 2: Reset Communication*

### 6.1.2 Reset Application

If an indication function is registered (see registerEvent_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.

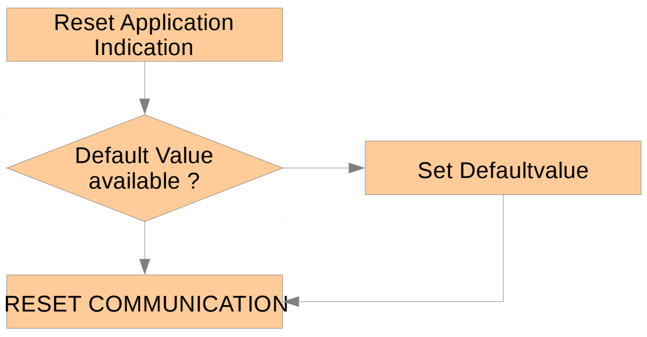


*Illustration 3: Reset Application*

### 6.1.3 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

## *6.2 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.

### 6.2.1 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 *coInitCanOpenStack().*

The indication function is called after each Reset Communication and Reset Application event and has to restore the parameters saved using Save Parameter. It can also be used to set hard-coded values if the objects 0x1010 (store parameters) and 0x1011 (restore parameters) are not present.

### 6.2.2 Save Parameter

Saving of object values into non-volatile memory is done after writing the special value 'save' into the object 0x1010. A corresponding function has to be registered using registerEvent_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 apllication 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.

### 6.2.3 Clear Parameter

Deleting the values stored in non-volatile memory is done after writing the special value 'kill' into the object 0x1011. A corresponding function has to be registered using registerEvent_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.

## 6.3 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.

### 6.3.1 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 registerEvent_SDO_SRV_READ, registerEvent_SDO_SRV_WRITE and registerEvent_SDO_SRV_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.

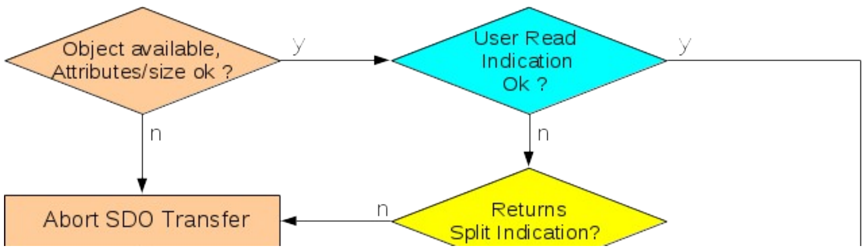


*Illustration 4: SDO Server Read*

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 toread/write data from an external (e.g. I²C) component.

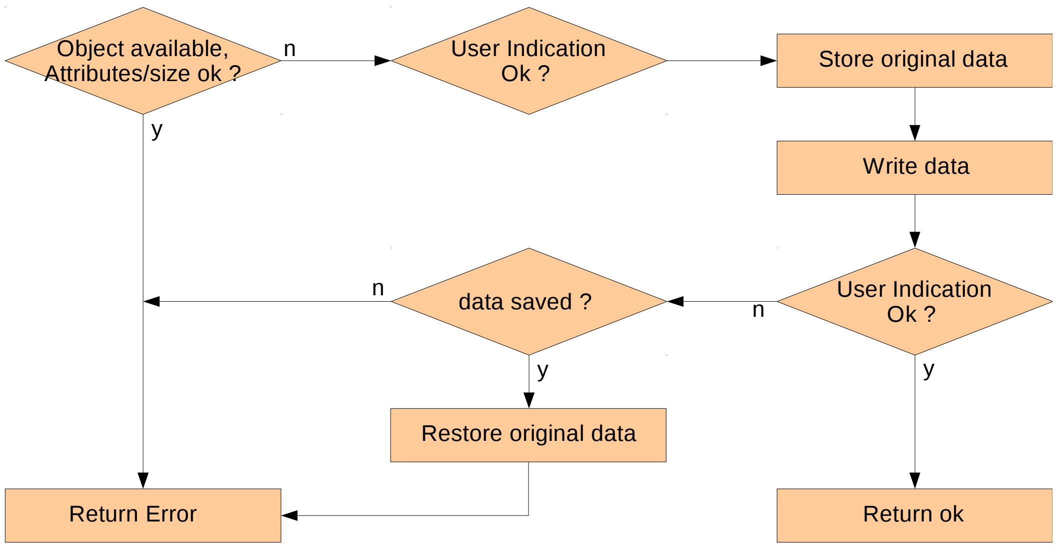


*Illustration 5: SDO Server Write*

### 6.3.2 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 registerEvent_SDO_CLIENT_READ and registerEvent_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.

### 6.3.3 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.

## 6.4 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 registerEvent_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 registerEvent_PDO and registerEvent_PDO_SYNC).

### 6.4.1 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) |

### 6.4.2 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 swaping) */
} 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)

### 6.4.3 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 registerEvent_PDO_REC_EVENT ).

### 6.4.4 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.

### 6.4.5 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 receivce PDOs are entered to the object dictionary.

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

### 6.4.6 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.

#### *6.4.6.1 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 |

##### 6.4.6.1.1 MPDO DAM Producer

Entries in Object Dictionary

| Index | SubIndex | Description | Value |
|---|---|---|---|
| $18xx_h$ | | PDO Communication Parameter | |
| $1Axx_h$ | 0 | Number of mapping entries | 255 |
| $1Axx_h$ | 1 | Mapping entrie | Appl. |

#### 6.4.6.1.2 MPDO DAM Consumer

Entries in Object Dictionary

| Index | SubIndex | 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

### *6.4.6.2 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 |

#### 6.4.6.2.1 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 | SubIndex | 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 Liste |  |

The format of the scanner list is:

| MSB |  | LSB |
|---|---|---|
| Bit 31..24 | Bit 23..8 | Bit 7..0 |
| Block Size | Index | SubIndex |

#### 6.4.6.2.2 MPDO SAM Consumer

Entries in Object Dictionary

| Index | SubIndex | Description | Value |
|---|---|---|---|
| $14xx_h$ |  | PDO Communication Parameter |  |

| Index | SubIndex | Description | Value |
|---|---|---|---|
| $16xx_h$ | 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.

## *6.5 Emergency*

### 6.5.1 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 registerEvent_EMCY).

### 6.5.2 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 registerEvent_EMCY_CONSUMER).

## *6.6 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 registerEvent_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.

### 6.6.1 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 registerEvent_NMT).

### 6.6.2 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.

### 6.6.3 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 registerEvent_EMCY).

## 6.7 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 registerEvent_SYNC and registerEvent_SYNC_FINISHED):

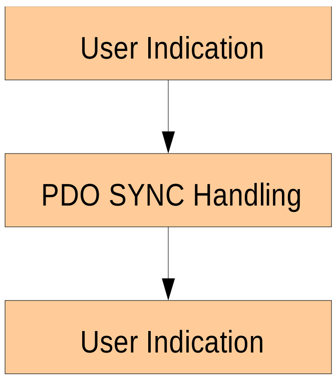


*Abbildung 6: SYNC Handling*

## 6.8 Heartbeat

### 6.8.1 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.

### 6.8.2 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 registerEvent_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 |

### 6.8.3 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 6.6.3Default Error Behavior) end a registered indication function is called (see registerEvent_ERRCTRL).

## 6.9 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 registerEvent_TIME).

## 6.10 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 LEDs can be switched on or off by a registered indication function (see registerEvent_LED).

## 6.11 LSS Slave

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

| **Status** | **Bedeutung** |
|---|---|
| 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 porvides 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 tovide 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 registerd 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.

## 6.12 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 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 registerEvent_CONFIG. The indication will inform if the transfer was successful or not.

The configuration itself is using SDO transfer. At least one SDO client hast to be configured on the NMT master. 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.

## 6.13 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 heatbeat monitoring for the active master. If the active master is lost, a new Flying Master Negotiation is started automatically.

## 6.14 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 registerEvent_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, | Transmit Queue is full, the message was not saved |

| | | |
|---|---|---|
| | Tr-Queue overflow | |
| TR-Queue empty | Bus-On, Tr-Queue ready | Transmit is ready to store messages (at least 50% free) |

## 6.15 *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.

## 6.16 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 bootup 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 *oEventRegister_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 bootup 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 |

# 7 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 timertick and the timertick 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 timertick only the first timer has to be checked as the following timers cannot be expired yet.

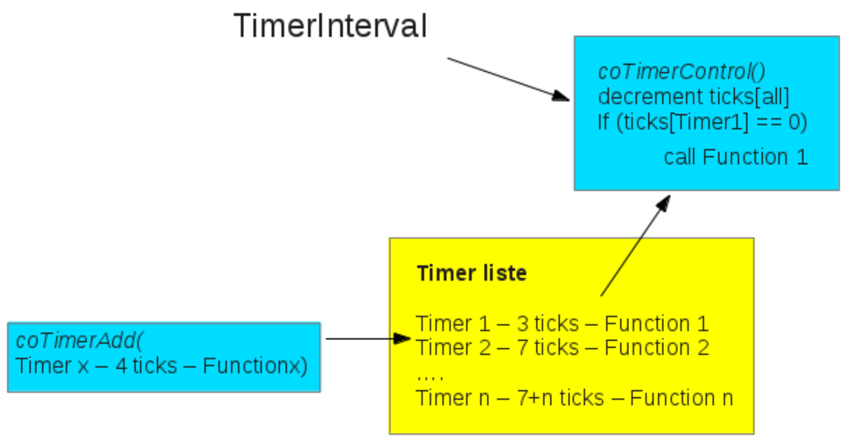

TimerInterval
coTimerControl()
decrement ticks[all]
If (ticks[Timer1] == 0)
call Function 1
Timer liste
Timer 1 – 3 ticks – Function 1
Timer 2 – 7 ticks – Function 2
....
Timer n – 7+n ticks – Function n
coTimerAdd(
Timer x – 4 ticks – Functionx)

*Illustration 7: 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().*

# 8 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.

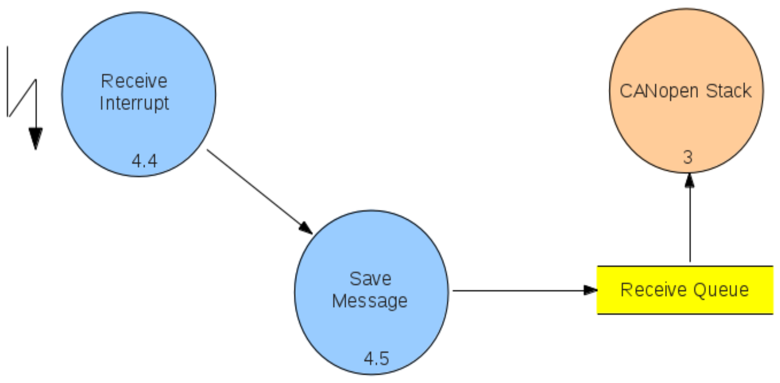
## *8.1 CAN Transmit*

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.

## 8.2 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.



# 9 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.

## 9.1 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)

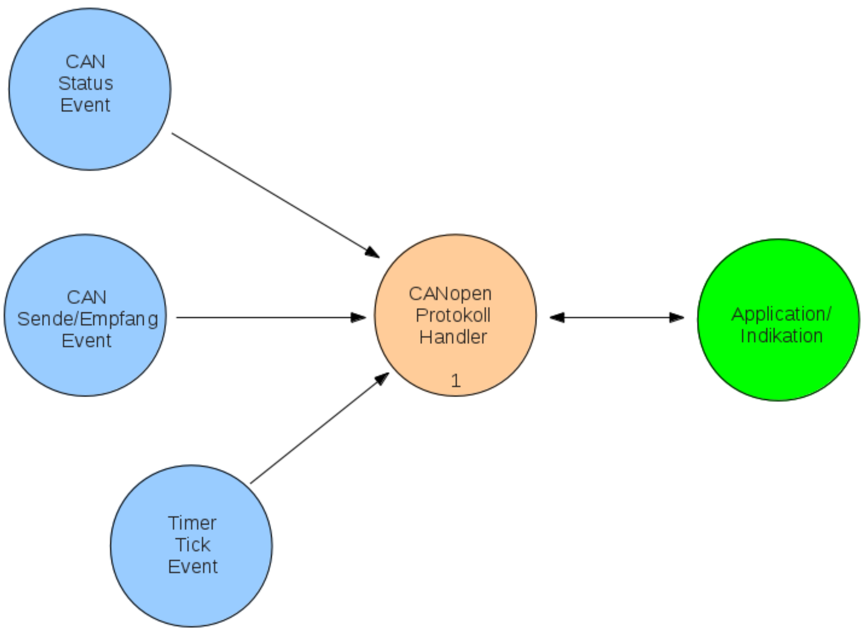


*Illustration 8: Process Signal Handling*

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 |

## 9.2 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:

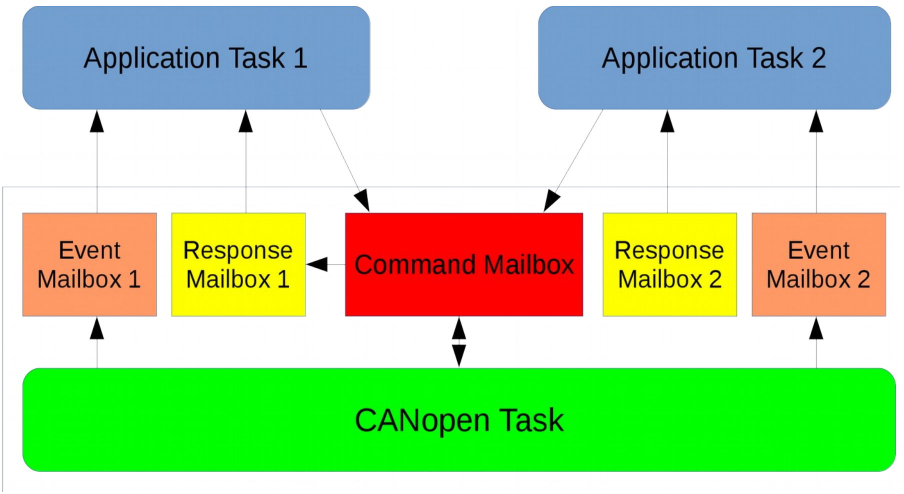
| | |
|---|---|
| CO_OS_LOCK_OD | Lock of the object dictionary |
| CO_OS_UNLOCK_OD | Unlock of the object dictionary |

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.

## 9.3 *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[2]. Any arbitrary number of application threads can be created that can send commands to the CANopen thread via a message queue.



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.

### 9.3.1 Creation of an application thread

Each application thread consists of a 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");
```

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

```
        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); };
```

### 9.3.2 Sending commands

For all basic CANopen functions and important CANopen master functions mailbox commands are available. To send such a command the corresponding structs 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 |

| CANopen function | Command |
|---|---|
| *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).

## 9.3.3 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 fuctions.

```
    /* wait for new events for oms*/
     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;
             }
     }
```

# 10 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.

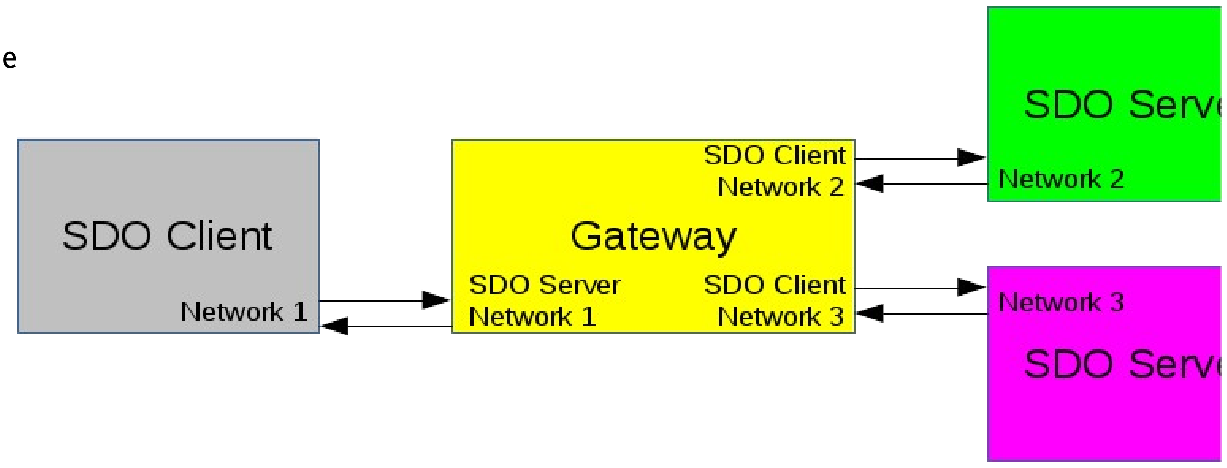
Examples for multi-line applications can be found in example/ml_*xxx*.

## 11 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.

### *11.1 SDO 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.

### *11.2 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.

## 12 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 examples/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/src (CANopen Stack)
   - Files in colib/inc (CANopen Stack public Header)
   - Files in example/slave1 (example application)
   - Files incodrv/<drivername> (driver)
4. Set include paths
   - examples/slave1
   - colib/inc
   - codrv/<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.

**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* |

# 13 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 methos 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.

# 14 Step by Step Guide – using CANopen Services

## 14.1 *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.

## 14.2 *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 an 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

## 14.3 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

## 14.4 Emergency Producer

Configuration using the CANopen DeviceDesigner:

- Create the Emergency Producer object 0x1014 using the CANopen DeviceDesigner
- Create the Error History object 0x1003 with n sub-indicies 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

## 14.5 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

## 14.6 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

## 14.7 PDOs

### 14.7.1 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 (see14.6)
    - 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

### 14.7.2 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 (see14.6)
    - 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

## 14.8 Dynamic objects

Activation in CANopen DeviceDesigner:

- Optional Services → Use Dynamic Objects

Configuration of the application:

- Initialization of dynamic object dictionary *coDynOdInit()*
- Add a 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

## 14.9 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

## 14.10 Configuration Manager

Configuration using the :

– Create objects 0x1F22 and 0x1F23 (Consive 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 a indication function using *registerEvent_CONFIG()*

– Put Concise DCF files into the object 0x1F22

– or read DCF file and convert them do concise DCF using *co_cfgConvToConcise()*

- 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.

# 15 Directory structure

| | |
|---|---|
| colib/src | CANopen Protocol Stack sources and private header files |
| colib/inc | CANopen Protocol Stack public header files |
| | (co_canopen.h includes all public headers) |
| codrv/ | CAN and CPU driver files |
| examples | example projects |
| examples/slave1 | |

.....

## 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 |