

# Anwenderhandbuch

## CANopen Master/Slave Protokoll Stack

V 2.6.4

### Versionshistorie

Version	Änderungen	Datum	Bearbeiter	Freigabe
0.9	Erste Version	14.06.2012	boe	
0.9.4	Anpassung an aktuellen Stack	24.08.2012	boe	
1.0	Step by Step Anleitung	22.11.2012	boe	
1.0.1	Add Store/Restore	06.12.2012	boe	
1.0.2	Dynamische Objekte	20.12.2012	boe	
1.1.0	Anpassung an Lib Version	09.03.2013	boe	
1.2.0	Anpassung an Lib Version	04.04.2013	boe	
1.3.0	Sleep Mode hinzugefügt	06.06.2013	boe	

Version	Änderungen	Datum	Bearbeiter	Freigabe
1.4.0	Add SDO Block transfer	08.07.2013	boe	
1.5.0	Add Objekt Indikation Handling	02.10.2013	boe	
1.6.0	Neue Features eingetragen	20.01.2014	boe	
1.7.0	Limit Check eingetragen	09.05.2014	boe	
1.8.0	Add U24..U64 Datentypen	15.06.2014	boe	
1.8.1	Dynamische Objekte, Mailbox API	10.09.2014	ged	
1.10.0	Statische Indikation Funktionen	03.11.2014	boe	
2.0.0	Add Multiline Kapitel	15.11.2014	boe	
2.2.0	Dynamische Objekte, Network Gateway	15.05.2015	boe	
2.2.3	Hinweise für Domain Indikation	16.06.2015	boe	
2.2.4	Bootup Procudure	20.06.2015	boe	
2.3.1	Split Indication/DynOd Applikation	05.07.2015	boe	
2.4.0	Add MPDO usage	10.08.2015	boe	
2.4.4	C#-Wrapper	30.10.2015	ged	
2.6.0	LSS Infos	23.05.2016	boe	
2.6.1	Store Funktionen hinzugefügt	14.06.2016	boe	
2.6.4	Add SDO Client Domain Indikation	23.09.2016	boe	

## Inhaltsverzeichnis

1 Übersicht.....	6
2 Eigenschaften.....	6
3 CANopen Protokoll Stack Konzept.....	7
4 Indikation Funktionen.....	9
5 Das Objektverzeichnis.....	12
5.1 Objektverzeichnis Variablen.....	12
5.2 Objekt Beschreibung.....	13
5.3 Objektverzeichnis Zuordnung.....	13
5.4 Strings und Domains.....	14
5.4.1 Domain Indication.....	14
5.5 Dynamisches Objektverzeichnis.....	14
5.5.1 Verwaltung mit Stackfunktionen.....	14
5.5.2 Verwaltung durch die Applikation.....	15
6 CANopen Protokoll Stack Dienste.....	15
6.1 Initialisierungsfunktionen.....	15
6.1.1 Reset Communication.....	15
6.1.2 Reset Applikation.....	16
6.1.3 Setzen der Knotennummer.....	16
6.2 Store/Restore.....	17
6.2.1 Load Parameter.....	17
6.2.2 Save Parameter.....	17
6.2.3 Clear Parameter.....	17
6.3 SDO.....	17
6.3.1 SDO Server.....	18
6.3.2 SDO Client.....	19
6.3.3 SDO Blocktransfer.....	20
6.3.4 SDO Client Network Requests.....	20
6.4 PDO.....	20
6.4.1 PDO Request.....	20
6.4.2 PDO Mapping.....	21
6.4.3 PDO Event Timer.....	21
6.4.4 RTR Handling.....	21
6.4.5 PDO und SYNC.....	22
6.4.6 Multiplexed PDOs (MPDOs).....	22
6.4.6.1 MPDO Destination Address Mode (DAM).....	23
6.4.6.1.1 MPDO DAM Producer.....	23
6.4.6.1.2 MPDO DAM Consumer.....	23
6.4.6.2 MPDO Source Address Mode (SAM).....	23
6.4.6.2.1 MPDO SAM Producer.....	23
6.4.6.2.2 MPDO SAM Consumer.....	24
6.5 Emergency.....	24
6.5.1 Emergency Producer.....	24
6.5.2 Emergency Consumer.....	24
6.6 NMT.....	25
6.6.1 NMT Slave.....	25

6.6.2 NMT Master.....	25
6.6.3 Default Error Behaviour.....	25
6.7 SYNC.....	25
6.8 Heartbeat.....	26
6.8.1 Heartbeat Producer.....	26
6.8.2 Heartbeat Consumer.....	26
6.8.3 Life Guarding.....	26
6.9 Time.....	27
6.10 LED.....	27
6.11 LSS Slave.....	27
6.12 Configuration Manager.....	28
6.13 Flying Master.....	28
6.14 Kommunikations-Status Auswertung.....	29
6.15 Sleep Mode für CiA 447 und CiA 454.....	29
6.16 Startup Manager.....	30
7 Timer Handling.....	31
8 Treiber.....	31
8.1 CAN Transmit.....	32
8.2 CAN Receive.....	32
9 Einbindung mit Betriebssystemen.....	33
9.1 Aufteilung in mehrere Tasks.....	33
9.2 Objektverzeichniszugriff.....	34
9.3 Mailbox-API.....	35
9.3.1 Einrichtung eines Applikationsthreads.....	35
9.3.2 Senden von Kommandos.....	36
9.3.3 Empfang von Events.....	37
10 Multi-Line Handling.....	38
11 Multi-Level Networking – Gateway Funktionalität.....	38
11.1 SDO Networking.....	38
11.2 EMCY Networking.....	39
11.3 PDO Forwarding.....	39
12 Beispiel Implementierung.....	39
13 C#-Wrapper.....	40
14 Dienste Schritt für Schritt.....	41
14.1 SDO Server Nutzung.....	41
14.2 SDO Client Nutzung.....	41
14.3 Heartbeat Consumer.....	42
14.4 Emergency Producer.....	42
14.5 Emergency Consumer.....	43
14.6 SYNC Producer/Consumer.....	43
14.7 PDOs.....	43
14.7.1 Empfangs-PDOs.....	43
14.7.2 Sende-PDOs.....	44
14.8 Dynamische Objekte.....	45
14.9 Objekt Indikation.....	45
14.10 Configuration Manager.....	45
15 Aufbau der Verzeichnisstruktur.....	46

## Referenzen

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 Übersicht

Der CANopen Slave Protokoll Stack stellt grundlegende Kommunikationsmechanismen für eine CANopen konforme Kommunikation von Geräten bereit und ermöglicht so Anwendern eine einfache und schnelle Integration von CANopen Kommunikationsdiensten in ihre Geräte. Dabei werden alle Dienste des CiA 301 bereitgestellt, die je nach Ausbaustufe in verschiedenen Modulen verfügbar sind, und über ein anwenderfreundliches User-Interface verfügen.

Für die einfache Portierbarkeit auf neue Hardwareplattformen ist der Protokoll Stack in einen hardware-unabhängigen und einen hardware-abhängigen Teil mit definiertem Interface aufgeteilt.

Die Konfiguration, Parametrierung und Skalierung erfolgt bei allen Diensten über ein grafisches Tool, um so optimalen Code und Laufzeiteffizienz zu ermöglichen.

## 2 Eigenschaften

- Trennung zwischen hardware- abhängigem/unabhängigem Teil mit definierten Interface
- ANSI-C konform
- Einhaltung der MISRA mandatory Regeln
- Unterstützung aller Dienste des CiA-301
- konform zu CiA-301 V4.2
- konfigurierbar und skalierbar
- Möglichkeiten für Erweiterungsmodule, besonders für Master-Funktionalitäten
- flexibles User-Interface
- statisches und dynamisches Objektverzeichnis
- mehrere Ausbaustufen
- LED CiA-303

CANopen Dienste der Ausbaustufen:

Dienstmerkmal	Basic Slave	Master/Slave	Manager
SDO Server	2	128	128
SDO Client		128	128
SDO expedited/segmented/block	●/●/-	●/●/○	●/●/○
PDO Producer	32	512	512
PDO Consumer	32	512	512
PDO Mapping	statisch	statisch/dynamisch	statisch/dynamisch
MPDO Destination Mode		○	○
MPDO Source Mode		○	○

Dienstmerkmal	Basic Slave	Master/Slave	Manager
SYNC Producer		●	●
SYNC Consumer	●	●	●
Time Producer		●	●
Time Consumer		●	●
Emergency Producer	●	●	●
Emergency Consumer		127	127
Guarding Master			●
Guarding Slave	●	●	●
Bootup Handling		●	●
Heartbeat Producer	●	●	●
Heartbeat Consumer		127	127
NMT Master-Funktionalität		●	●
NMT Slave	●	●	●
LED CiA-303	●	●	●
LSS CiA-305	●	●	●
Sleep Mode nach CiA-454	●	●	●
Master Bootup CiA-302			●
Configuration Manager			●
Flying Master		○	○
Redundanz		○	○
Safety (SRD0)	○	○	○
Multiline		○	○
CiA-4xx	○	○	○

● - inklusive, ○ - optional

### 3 CANopen Protokoll Stack Konzept

- Alle Dienste und Funktionalitäten sind per #define Anweisungen ein-/ausschaltbar
- die Konfiguration erfolgt über das Konfigurationstool CANopen DeviceDesigner
- strikte Datenkapselung, Zugriff erfolgt nur über Funktionsaufrufe bei unterschiedlichen Modulen (keine globalen Variablen)
- Jeder Dienst stellt eine eigene Initialisierungsfunktion zur Verfügung

Die

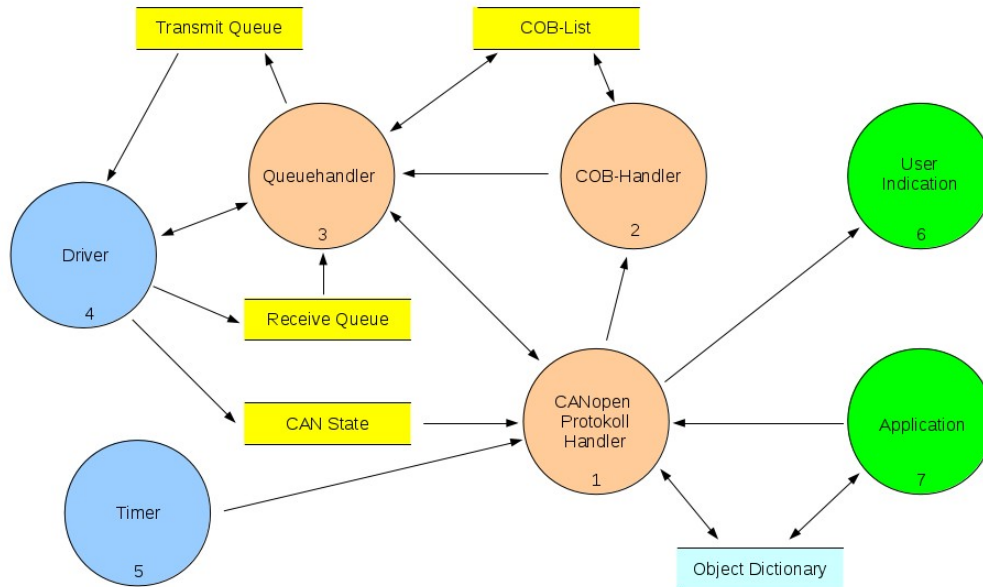


Abbildung 1: Überblick über die Module

Funktionsblöcke (FB)

- CANopen Protokoll Handler (FB 1)
- COB-Handler (FB 2)
- Queue-Handler (FB 3)
- Treiber (FB 4)

werden von der zentralen Bearbeitungsfunktion *coCommTask()* aufgerufen, damit alle CANopen Funktionen ausgeführt werden können.

Diese zentrale Bearbeitungsfunktion ist aufzurufen wenn:

- neue Nachrichten in der Empfangsqueue verfügbar sind
- die Timerperiode abgelaufen ist
- der CAN/Kommunikations-Status sich geändert hat.

Wenn ein Betriebssystem vorhanden ist, kann dies sehr leicht über Signale angezeigt werden. Im Embedded Bereich ist aber auch ein Pollen der Funktion möglich.

Funktionsaufrufe für CANopen Dienste liefern standardmäßig den Ausführungsstatus als Datentyp *RET\_T* zurück. Bei Funktionen mit Anfragen an andere Knoten ist der Rückgabewert nicht die Antwort des angefragten Knotens, sondern der Status der Anfrage. Die Antwort des angefragten Knotens wird dann über eine Indikation Funktion geliefert. Indikation Funktionen müssen vorher angemeldet werden (siehe Kapitel 4).



## 4 Indikation Funktionen

Interne Events im CANopen Protokoll Stack können mit einer User-Indikation verknüpft werden. Dafür muss die Applikation eine entsprechende Funktion bereit stellen, die bei dem entsprechenden Ereignis aufgerufen wird. Events können mit der folgenden Funktion angemeldet werden:

```
coEventRegister_<EVENT_TYPE>(&functionName);
```

Für jedes Event können auch mehrere Funktionen registriert werden, die dann nacheinander aufgerufen werden. Die Anzahl ist mit dem CANopen Device Designer festzulegen.

Wenn ein Event nicht angemeldet werden konnte (z.B. Dienst nicht verfügbar), wird eine entsprechende Fehlermeldung zurückgeliefert. Der Datentyp für *functionName-Pointer* ist vom jeweiligen Dienst abhängig.

Folgende Events können angemeldet werden:

EVENT_TYPE	Event	Parameter	Rückgabewert
COMM_EVENT	Kommunikationsstatus	CAN/Komm-Status	
CAN_STATE	CAN Status	CAN Status	
EMCY	automatisch generierte Emergency Nachricht soll versendet werden	Fehlercode Zeiger auf Addit. Bytes	Emcy senden/nicht senden
EMCY_CONSUMER	Emergency Consumer Nachricht erhalten	Knoten Nummer Fehlercode Fehlerregister Additional Bytes	
LED_GREEN/LED_RED	Grüne/Rote Led setzen	ein/aus	
ERRCTRL	Heartbeat/Bootup Status	Knoten Nummer HB Status NMT Status	
NMT	NMT Status Wechsel	Neuer NMT Status	Ok/nicht Ok
LSS	LSS slave information	Service bitrate Zeiger für ErrorCode Zeiger für ErrorSpec	Ok/Nicht ok
LSS_MASTER		Servicenummer ErrorCode ErrorSpec Zeiger auf Identity	
PDO	asynchrones PDO empfangen	PDO Nummer	
PDO_SYNC	synchrones PDO empfangen	PDO Nummer	
PDO_REC_EVENT	Time Out für PDO	PDO Nummer	
MPDO	Multiplexed PDO empfangen	PDO Nummer Index	

EVENT_TYPE	Event	Parameter	Rückgabewert
		Subindex	
SDO_SERVER_READ	SDO Server Read Transfer beginnt	SDO Server Nummer index subindex	Ok/SDO abort code/ Split Indikation
SDO_SERVER_WRITE	SDO Server Write Transfer beendet	SDO Server Nummer index subindex	Ok/SDO abort code/ Split Indikation
SDO_SERVER_CHECK_WRITE	SDO Server Write Transfer beginnt	SDO Server Nummer index subindex pointer to received data	Ok/SDO abort code
SDO_SERVER_DOMAIN_WRITE	SDO Domain Größe erreicht	Index subindex Domain Buffer Size Transferred Size	
SDO_CLIENT_READ	SDO Client Read Transfer beendet	SDO Client Nummer index subindex Anzahl Daten result	
SDO_CLIENT_WRITE	SDO Client Write Transfer beendet	SDO Client Nummer index subindex result	
OBJECT_CHANGED	Objekt wurde durch SDO oder PDO Zugriff geändert	index subIndex	Ok/SDO abort Code
SYNC	SYNC Nachricht empfangen		
SYNC_FINISHED	SYNC Bearbeitung abgeschlossen		
TIME	Time Nachricht erhalten	Zeiger auf Timestruktur	
LOAD_PARA	Gespeicherte Objekte restaurieren	Subindex/OD-Bereich	
SAVE_PARA	Objekte nichtflüchtig speichern	Subindex/OD-Bereich	
CLEAR_PARA	Gespeicherte Objekte löschen	Subindex/OD-Bereich	
SLEEP	Sleep Mode State	Sleep Mode State	Ok/Abort
CFG_MANAGER	DCF Schreiben beendet	Transfer index subindex reason	
FLYMA	Flying Master Status	State Master Node Priorität	
SRD	SRD Antwort vom Master	Result	

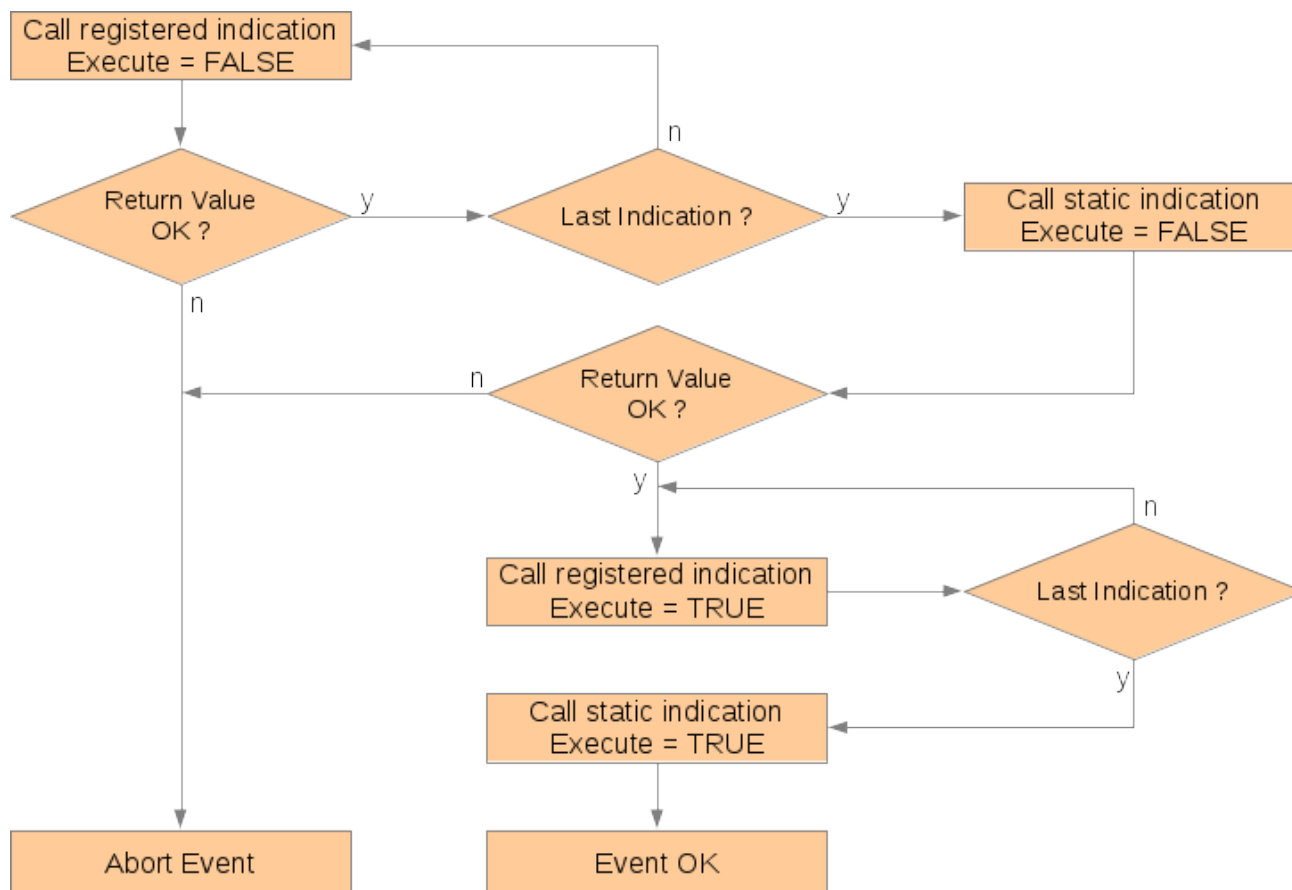
EVENT_TYPE	Event	Parameter	Rückgabewert
		Fehlercode	
GW_SDOCLIENT_USER	Client SDO für Gateway Funktionalität	-	SDO Nr

Für jeden dieser Events kann auch eine statische Indikation Funktion zur Compile-Zeit festgelegt und eingebunden werden. Statische Indikation Funktionen werden immer nach den dynamisch festgelegten Funktionen im Stack aufgerufen.

Alle Indikation Funktionen die eine Rückgabewert liefern, verfügen über einen zusätzlichen Parameter:

Parameter	Wert	Bedeutung
execute	CO_FALSE	Rückgabewert der Funktion wird ausgewertet. Indikation Funktionalität darf noch <b>nicht</b> ausgeführt werden
	CO_TRUE	Rückgabewert der Funktion wird nicht ausgewertet. Indikation Funktionalität soll ausgeführt werden

Bei einem entsprechenden Event werden alle dafür registrierten Indikation Funktionen mit dem Parameter execute = CO\_FALSE aufgerufen. Dabei soll in den Indikation Funktionen geprüft werden, ob das Event ausgeführt werden darf oder nicht. Nur wenn alle Funktionen ein RET\_OK liefern, werden anschließend alle Indikation Funktionen nochmals mit dem Parameter execute = CO\_TRUE aufgerufen, damit die Funktionalitäten ausgeführt werden können.



## 5 Das Objektverzeichnis

Das Objektverzeichnis wird mit dem CANopen DeviceDesigner generiert und während der Initialisierung dem CANopen Protokoll Stack übergeben. Dabei gelten die vom CiA 301 festgelegten Eigenschaften, die auch Lücken in den Subindizes erlauben.

Alle Objekte im Kommunikationsbereich (1000h-1fffh) werden direkt durch die entsprechenden Dienste verwaltet. Der Zugriff auf diese Objekte kann nur über Funktionsaufrufe erfolgen. Alle anderen Objekte können als:

- verwaltete Variablen (Variable wird vom Stack verwaltet)
- konstante Variablen (Konstante wird vom Stack verwaltet)
- als Zeiger auf Variable in der Applikation

angelegt werden.

Für die vom Stack verwalteten Variablen und Konstanten stehen die Zugriffsfunktionen [coOdGetObj\\_xx](#) und [coOdPutObj\\_xx](#) für die jeweiligen Datentypen zur Verfügung.

Weitere Objekteigenschaften wie Zugriffsrechte, Größeninformation oder Defaultwerte können mit den Funktionen [coOdGetObjAttribute\(\)](#), [coOdGetObjSize\(\)](#) bzw. [coOdGetDefaultVal\\_xx](#) ermittelt werden.

Für das vereinfachte Setzen von COB-IDs steht die Funktion [coOdSetCobid\(\)](#) zur Verfügung.

Die Objektverzeichnis Implementierung besteht aus 3 Teilen:

- Variablen (im RAM, ROM oder über Zeiger)
- Subindex Beschreibung
- Objektverzeichnis Zuordnung Index, Subindexbeschreibung

### 5.1 Objektverzeichnis Variablen

Für jeden Variablentyp können bis zu 3 Arrays angelegt werden:

#### Verwaltete Variablen:

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

#### Konstante Variablen

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

#### Zeiger auf Variablen

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

Die Definition und Verwaltung der Arrays erfolgt durch den CANopen DeviceDesigner.

## 5.2 Objekt Beschreibung

Die Objekt Beschreibung ist für jeden Subindex vorhanden. Sie enthält folgende Informationen:

Information	Bedeutung
subindex	Subindex des Objekts
dType	Datentyp des Objekts (var, const, pointer, service)
tableIdx	Index in der jeweiligen Tabelle von dType oder Servicenummer
attr	Objekt Attribute
defValIdx	Index in der konstanten Tabelle für den Default Wert
limitMinIdx	Index in der konstanten Tabelle für den minimalen Grenzwert
limitMaxIdx	Index in der konstanten Tabelle für den maximalen Grenzwert

Definition der Attribute:

CO_ATTR_READ	Objekt lesbar
CO_ATTR_WRITE	Objekt schreibbar
CO_ATTR_NUM	Objekt ist numerisch
CO_ATTR_MAP_TR	Objekt ist in TPDO mapbar
CO_ATTR_MAP_REC	Objekt ist in RPDO mapbar
CO_ATTR_DEFVAL	Objekt hat Default Werte
CO_ATTR_LIMIT	Objekt hat Grenzwerte

Der Limit-Check für Objekte kann für jedes Objekt individuell mit Hilfe des CANopen DeviceDesigners eingetragen werden.

## 5.3 Objektverzeichnis Zuordnung

Die Objektverzeichnis Zuordnung ist für jeden Index einmal vorhanden. Sie enthält:

index	Objekt Index
numberOfSubs	Anzahl der Subindexes
highestSub	Höchster genutzter Subindex
odType	Datentyp (Variable, Array, Record)
odDescIdx	Index in der object_description table

## 5.4 Strings und Domains

Strings werden unterschiedlich behandelt:

- konstante Strings werden im Objektverzeichnis verwaltet. Dazu existiert eine Liste mit den Zeigern auf die Strings und parallel dazu eine Liste mit den Größeninformationen. Beide Listen sind konstant und nicht änderbar.
- variable Strings werden wie Domains behandelt und in der Domainliste eingetragen.

Für Domains können Adressen und Größe zur Laufzeit mit der Funktion [coOdDomainAddrSet\(\)](#) eingestellt werden. In einer Domainliste werden die Adressen der jeweiligen Domain verwaltet, parallel dazu existiert eine Liste mit den Größeninformation.

### 5.4.1 Domain Indication

Domains können beliebige Größen annehmen und auch für z.B. Programmdownloads genutzt werden. In diesem Fall können in der Regel nicht alle Daten im RAM zwischengespeichert werden, sondern müssen nach Erreichen einer bestimmten Puffergröße z.B. in den Flash geschrieben werden. Dafür kann die Indikation Funktion [coEventRegister\\_SDO\\_SERVER\\_DOMAIN\\_WRITE\(\)](#) genutzt werden. Die angemeldete Indikation Funktion wird nach Erreichen einer vorgegebenen Anzahl von CAN Nachrichten aufgerufen, so dass die Daten der Domain in den Flash geschrieben werden können. Der zugehörige Domainpuffer wird anschließend gelöscht und erneut vom Anfang beschrieben.

Achtung!! Das Verhalten gilt für alle Domain Objekte. Die angegebene Größe der vorgegebenen CAN Nachrichten und damit das Rücksetzen des Puffers erfolgt somit immer, sobald die Nachrichtengröße erreicht ist. Falls andere und größere Domains genutzt werden sollen, müssen die Daten ggf. umkopiert werden.

## 5.5 Dynamisches Objektverzeichnis

### 5.5.1 Verwaltung mit Stackfunktionen

Objekte im Herstellerspezifischen- und Geräteprofil-Bereich können auch dynamisch zur Laufzeit angelegt werden. Damit können im Programm vorhandene oder auch dynamisch angelegte Variablen über das Objektverzeichnis zugänglich gemacht werden. Es erfolgt somit eine Verknüpfung von Variable und Objektverzeichnis-Index und SubIndex. Dynamische Objekte können mit dem Datentyp INTEGER8, INTEGER16, INTEGER32, UNSIGNED8, UNSIGNED16, UNSIGNED32 oder UNSIGNED64 angelegt werden.

Für die Verwaltung dieser Objekte wird vom Stack dynamischer Speicher angefordert. Dies erfolgt über die Funktion [coDynOdInit\(\)](#), der die maximale Anzahl der zu verwendenden Objekte zu übergeben sind.

Objekte werden mit der Funktion [coDynOdAddIndex\(\)](#) angelegt, zugehörige Subindizes über die Funktion [coDynOdAddSubIndex\(\)](#). Dabei können auch die Eigenschaften wie Zugriffsrechte, PDO-mapbar, numerischer Wert,... festgelegt werden.

Der Zugriff und die Nutzung der dynamisch angelegten Objekte erfolgt mit den Standard Zugriffsfunktionen. Sie können daher auch bei allen Diensten wie PDO oder SDO ohne Einschränkungen genutzt werden.

Als Beispiel siehe examples/dynod.

## 5.5.2 Verwaltung durch die Applikation

Dynamische Objekte können auch direkt durch die Applikation angelegt und verwaltet werden. Dafür muss die Applikation die entsprechenden Funktionen zur Verfügung stellen:

```
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 */
```

Vom Stack wird immer zuerst die Objekt Beschreibung ermittelt, und damit dann die Adresse bzw. Größe des Objekts angefragt. Die von der Applikation verwalteten Objekte können auch in PDOs gemappt werden. In diesem Fall muss das Objekt aber immer verfügbar sein, da die Daten direkt über den ermittelten Zeiger auf das Objekt geschrieben werden.

Für die Nutzung kann das Beispiel unter `example/dynod_appl` genutzt werden.

Achtung! Die gleichzeitige Nutzung der dynamischen Objekte durch den Stack und die Applikation ist nicht möglich!

## 6 CANopen Protokoll Stack Dienste

### 6.1 Initialisierungsfunktionen

Vor der Nutzung des CANopen Protokoll Stacks sind folgende Initialisierungen vorzunehmen:

<code>coInitCanOpenStack()</code>	Initialisierung der CANopen Dienste und des Objektverzeichnisses
<code>codrvCanInit()</code>	Initialisierung des CAN Controllers
<code>codrvTimerSetup()</code>	Bereitstellung eines Timer-Intervals (z.B. Hardwaretimer)
<code>codrvCanEnable()</code>	Start des CAN Controllers

#### 6.1.1 Reset Communication

Rücksetzen der Kommunikationsvariablen (Index 0x1000..0x1fff) im Objektverzeichnis auf die Default Werte. Dabei werden die COB-IDs entsprechend dem Pre-Defined Connection Set gesetzt. Anschließend wird die registrierte Event Funktion (siehe `registerEvent_NMT`) aufgerufen.

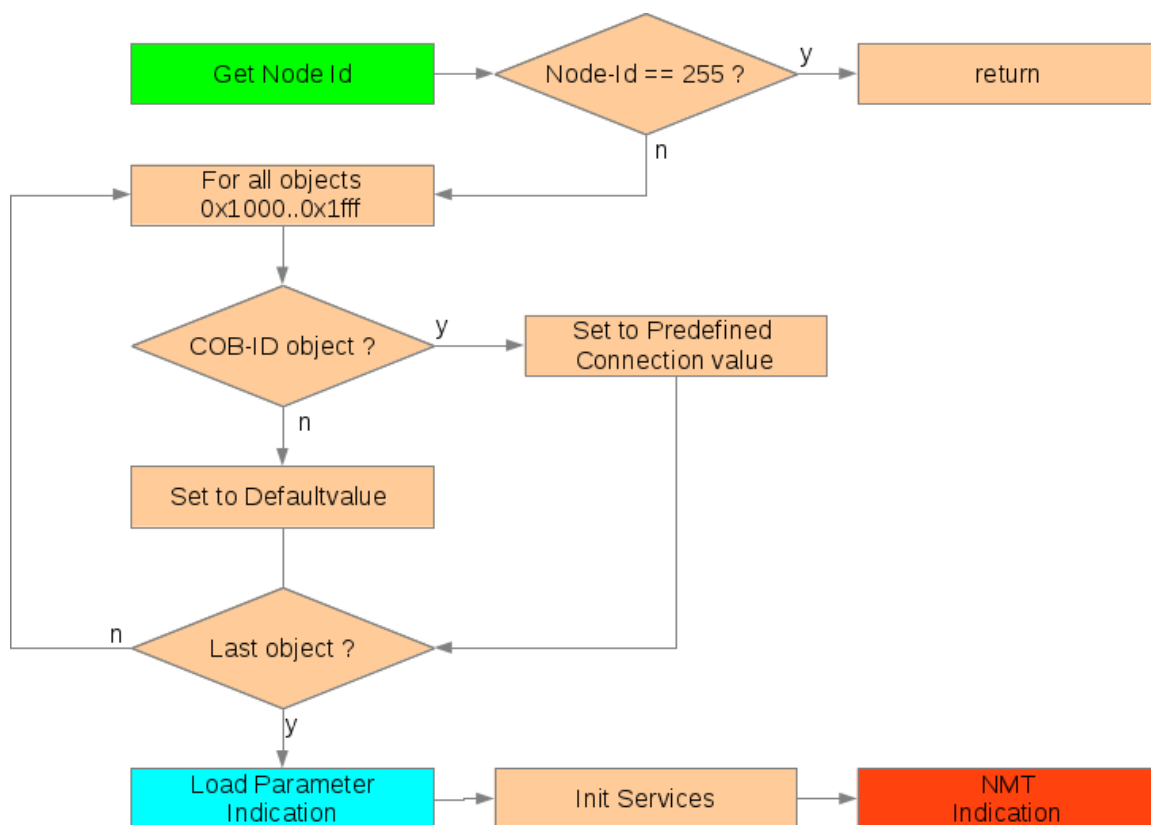


Abbildung 2: Reset Communication

### 6.1.2 Reset Applikation

Bei Bedarf kann als erstes die registrierte Event Funktion (siehe registerEvent\_NMT) aufgerufen werden, um ggf. Dinge in der Applikation auszuführen (z.B. Motor anhalten). Anschließend werden alle Objektvariablen auf ihre Defaultwerte gesetzt, und ein Reset Communication durchgeführt.

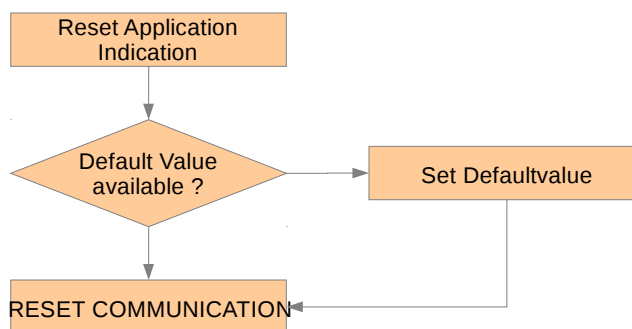


Abbildung 3: Reset Application

### 6.1.3 Setzen der Knotennummer

Die Knotennummer muss im Bereich von 1..127 bzw. 255 (Datentyp unsigned char) liegen und kann gesetzt werden über

- Konstante zur Compile-Zeit



- Variable
- Funktionsaufruf
- LSS

Dafür ist im CANopen DeviceDesigner das Eingabefeld entsprechend zu belegen.

Hinweise:

Für LSS ist die Knotennummer auf 255u zu setzen.

Wird die Knotennummer über einen Funktionsaufruf oder eine Variable bereitgestellt, sollte der Prototyp für die Funktion bzw. die Extern-Deklaration für die Variable auch im gen\_define.h definiert werden. Wenn beide definiert sind, wird die Funktion genutzt.

## 6.2 Store/Restore

Der Stack unterstützt die Store/Restore Funktionalität nur nach Aufforderung durch Schreiben auf die Objekte 0x1010 und 0x1011. Ein Lesen dieser Objekte liefert daher immer den Wert 1.

Für die nicht-flüchtige Speicherung bzw. Wiederherstellung ist die Applikation verantwortlich.

### 6.2.1 Load Parameter

Beim Aufruf der Reset-Funktionen (Reset Communication, Reset Application) können die Default-Werte für die Objekte über eine Indikation Funktion überschrieben werden. Die Anmeldung der Indikation Funktion kann automatisch durch die Initialisierungsfunktion [coInitCanOpenStack\(\)](#) erfolgen.

Die Indikation Funktion wird dann bei jedem Reset Communication und Reset Applikation aufgerufen, und soll die mit SAVE Parameter (siehe Abschnitt 16.2.2) gespeicherten Werte wiederherstellen.

Sie kann auch genutzt werden um fest kodierte Werte zu setzen, wenn die Objekte 0x1010 (store parameter) und 0x1011 (restore parameter) nicht vorhanden sind.

### 6.2.2 Save Parameter

Die Sicherung von Objekten in einem nicht-flüchtigen Speicher erfolgt nach dem Schreiben auf das Objekt 0x1010 mit dem speziellen Wert „save“. Dafür ist eine entsprechende Funktion über [registerEvent\\_SAVE\\_PARA\(\)](#) zu hinterlegen, die das Speichern ermöglicht.

Welche Objekte gesichert werden, kann applikationsspezifisch festgelegt werden. Der Stack bietet mit den Funktionen [odGetObjStoreFlagCnt\(\)](#) und [odGetObjStoreFlag\(\)](#) die Möglichkeit, die mit dem CANopen DeviceDesigner gekennzeichneten zu speichernden Objekte zu ermitteln.

### 6.2.3 Clear Parameter

Das Löschen der Objektverzeichnis Daten im nicht-flüchtigen Speicher erfolgt nach dem Schreiben auf das Objekt 0x1011 mit dem speziellen Wert „load“. Dafür ist eine entsprechende Funktion über [registerEvent\\_CLEAR\\_PARA\(\)](#) zu hinterlegen, die das Löschen in dem nicht-flüchtigen Speicher realisiert.

Bei einem folgenden Reset Applikation oder Reset Communication sollten dann beim Aufruf der Load Parameter Funktion (siehe 6.2.1) keine Daten mehr geladen werden.

## 6.3 SDO

Die COB-IDs für das erste Server SDO werden bei einem Reset Communication automatisch auf die Predefined Connection Set Werte gesetzt. Alle anderen COB-IDs von SDOs sind nach einem Reset

Communication disabled.

Generell können COB-IDs nur modifiziert werden, wenn das Disable-Bit in der COB-ID vorher gesetzt wurde.

### 6.3.1 SDO Server

SDO Server Dienste sind passiv. Sie werden durch Nachrichten von externen SDO Clients getriggert und reagieren nur entsprechend der eintreffenden Nachrichten. Die Applikation wird über Start- und Ende dieser Transfers durch die registrierten Event Funktionen (siehe `registerEvent_SDO_SERVER_READ`, `registerEvent_SDO_SERVER_WRITE` und `registerEvent_SDO_SERVER_CHECK_WRITE`) informiert.

Der SDO Server Handler wertet die empfangenen Daten aus. Dabei wird überprüft, ob die Objektverzeichnis Einträge verfügbar und die Zugriffsattribute korrekt sind. Anschließend werden die Daten im Objektverzeichnis hinterlegt bzw. ausgelesen. Vor bzw. nach der Übertragung können entsprechende User-Indikation-Funktionen aufgerufen und ausgewertet werden, die auch auf die Antwort vom Client Einfluss haben.

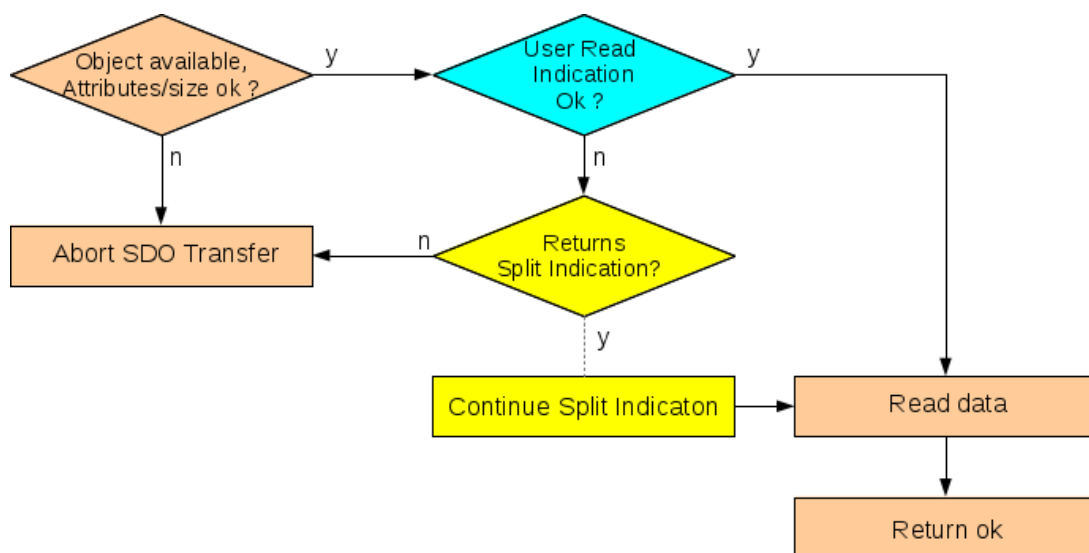


Abbildung 4: SDO Server Read

Die registrierten Event Funktionen können zusätzlich mit dem Parameter `RET_SDO_SPLIT_INDICATION` verlassen werden. In diesem Fall wird die weitere Bearbeitung der Nachricht und die Generierung der Antwort unterbrochen, bis die Funktion `coSdoServerReadIndCont()` bzw. `coSdoServerWriteIndCont()` aufgerufen wurde.

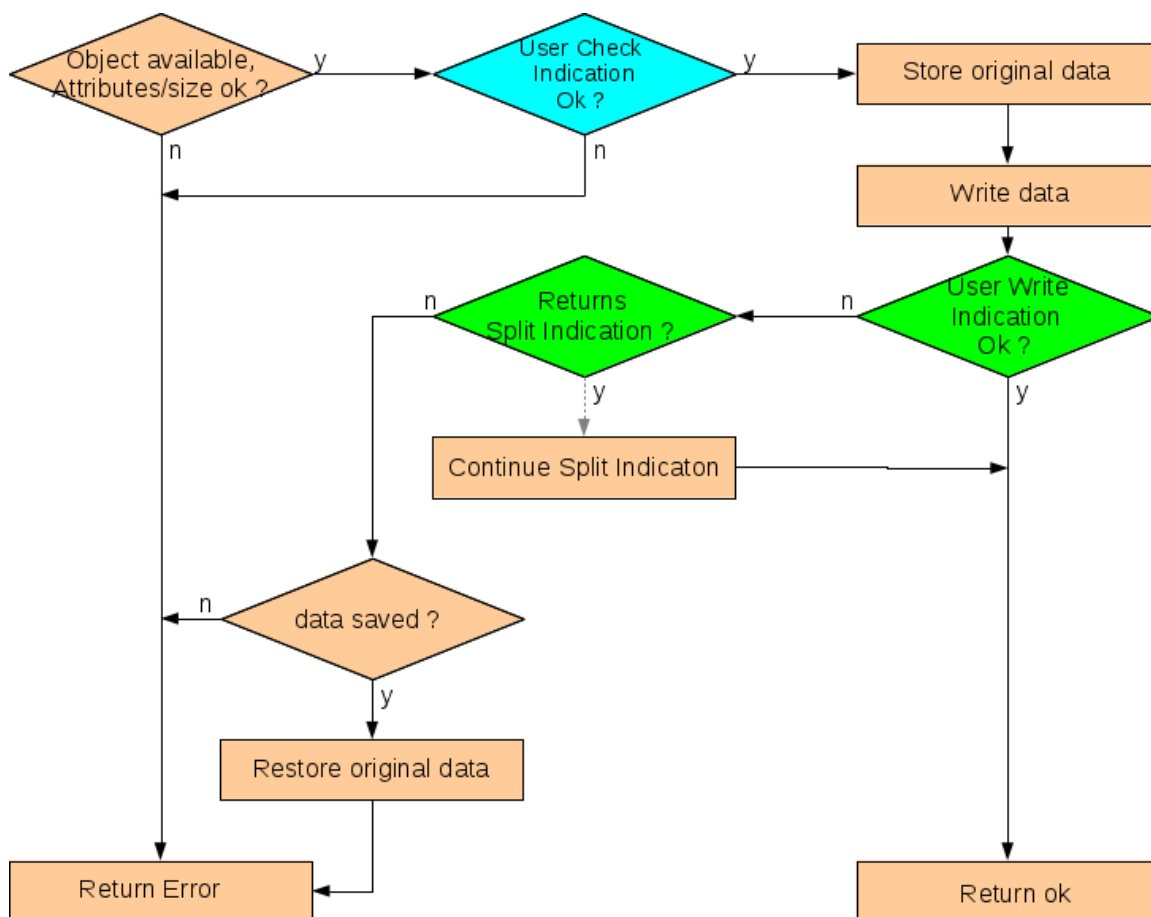
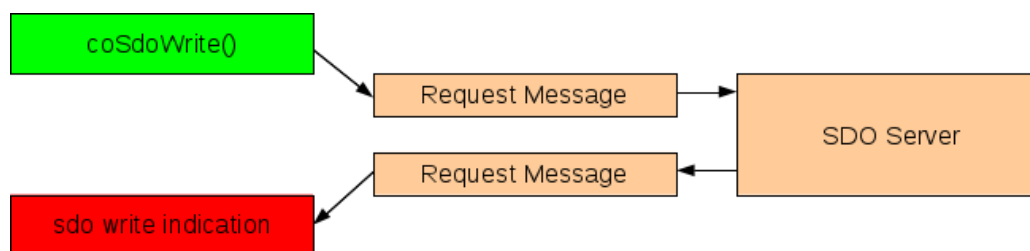


Abbildung 5: SDO Server Write

### 6.3.2 SDO Client

SDO Client Dienste müssen von der Applikation angefordert werden. Dafür stehen die Funktionen `coSdoRead()` und `coSdoWrite()` zur Verfügung. Sie starten den notwendigen Transfer. Asynchron dazu wird der Abschluss der Übertragung über die registrierten Event Funktionen (siehe `registerEvent_SDO_CLIENT_READ` und `registerEvent_SDO_CLIENT_WRITE`) mitgeteilt.

Bei jeder Nachrichtenübertragung wird eine Timeout-Überwachung gestartet, die nach Ablauf der eingestellten Zeit die registrierten Event Funktionen (siehe `registerEvent_SDO_CLIENT_READ` und `registerEvent_SDO_CLIENT_WRITE`) auslöst. Der Timeout-Wert gilt jeweils für ein Telegramm. Wenn die Übertragung aus mehreren Telegrammen besteht (segmentierter Transfer), dann wird diese Zeit für jedes Telegramm angewendet.



### 6.3.3 SDO Blocktransfer

Der SDO Blocktransfer wird beim SDO Client automatisch genutzt, sobald die Datengröße den im CANopen DeviceDesigner eingestellten Wert überschreitet. Unterstützt der Server kein Blocktransfer, wird auf segmentierten Transfer umgeschaltet und der Transfer wiederholt.

SDO Blockanfragen werden vom SDO Server immer als SDO Blocktransfer beantwortet.

Die CRC Berechnung ist optional und kann im CANopen DeviceDesigner aktiviert werden. Sie erfolgt über vorkonfigurierte Tabellen.

### 6.3.4 SDO Client Network Requests

SDO Client Network Requests erfolgen mit den Funktionen *coSdoNetworkRead()* und *coSdoNetworkWrite()* und werden analog den SDO Client Read und Client Write Aufrufen behandelt.

## 6.4 PDO

Das PDO Handling erfolgt automatisch. Dabei werden alle Daten entsprechend dem eingestellten Mapping in die vorgesehenen Objekte kopiert bzw. von dort geholt. Ebenso werden die Inhibit Berechnung oder die timergetriebenen und synchronen PDOs automatisch behandelt.

Beim Empfang von PDOs mit fehlerhafter Länge und eingeschaltetem Emergency-Dienst wird automatisch eine Emergency Nachricht versendet. Die 5 applikationsspezifischen Bytes der Emergency-Nachricht können über die registrierten Event Indikation Funktion (siehe registerEvent\_EMCY) modifiziert werden. Standardmäßig enthalten sie folgende Informationen:

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

Synchrone PDOs werden automatisch nach dem Eintreffen der SYNC-Nachricht aus den Objekten gemappt und versendet. Ebenfalls werden die nach dem letzten SYNC-Nachricht empfangenen PDOs in die gemappten Objekte geschrieben.

Jedes empfangene PDO kann über eine Indikation-Funktion gemeldet werden. Dabei können für synchrone und asynchrone PDOs jeweils eigene Event Funktionen registriert werden (siehe registerEvent\_PDO und registerEvent\_PDO\_SYNC).

### 6.4.1 PDO Request

Das Senden eines PDOs ist nur für asynchrone und synchron-azyklische PDOs erlaubt. Dafür stehen 2 Funktionen zur Verfügung:

- coPdoReqNr()*            PDO mit bestimmter PDO Nummer versenden
- coPdoReqObj()*        PDO versenden, das diesen Index und Subindex enthält

### 6.4.2 PDO Mapping

Das PDO Mapping erfolgt über Mappingtabellen. Diese werden bei statischen Mapping in konstanten Mappingtabellen vom CANopen DeviceDesigner erzeugt. Bei dynamischen Mapping werden die Mapping-Tabellen bei der Initialisierung und bei der Aktivierung des Mapping (Schreiben auf sub o) generiert.

Aufbau der Mappingtabelle:

```
typedef struct{
    void *pVar;          /* Zeiger auf die Variable */
    U8 len;              /* Anzahl der Bytes für diese Variable */
    FLAG_T numeric;      /* Kennzeichen numerisch für Byteswapping */
} PDO_MAP_ENTRY_T;

typedef struct{
    U8    mapCnt;        /* Anzahl der gemappten Variablen */
    PDO_MAP_ENTRY_T  mapEntry[]; /* Mapping Einträge */
} PDO_MAP_TABLE;
```

Beim Ändern des Mappings sind folgende Schritte notwendig:

- PDO ausschalten (NO\_VALID\_BIT in PDO COB-ID setzen)
- Mapping ausschalten (Subindex o der Mappingeinträge auf 0 setzen)
- Mapping Einträge modifizieren
- Mapping einschalten (Subindex o der Mappingeinträge auf gewünschte Anzahl setzen)
- PDO einschalten (NO\_VALID\_BIT in PDO COB-ID rücksetzen)

### 6.4.3 PDO Event Timer

PDO Event Timer können für Sende-PDOs im asynchronen, und für Empfangs-PDOs in allen Modes (außer RTR) genutzt werden. Bei Sende-PDOs wird nach Ablauf der eingestellten Event Time das PDO automatisch erneut versendet. Bei Empfangs-PDOs startet nach dem Eintreffen des jeweiligen PDOs eine Time-Out Überwachung mit der eingestellten Event Time. Wenn diese abgelaufen ist, bevor das PDO erneut empfangen wurde, wird die registrierte Indikation Funktion (siehe registerEvent [PDO\\_REC\\_EVENT](#)) aufgerufen.

### 6.4.4 RTR Handling

Wenn der Treiber bzw. die Hardware keine RTRs behandeln können, ist bei allen PDO COB-Ids das Bit 30 zu setzen (0x4000 0000). Mit dem define CO\_RTR\_NOT\_SUPPORTED wird das Rücksetzen dieses Bits verhindert.

### 6.4.5 PDO und SYNC

Mit dem SYNC Dienst kann sowohl die Datenübertragung als auch die Datenerfassung im Netzwerk synchronisiert werden. Nach dem Eintreffen bzw. Senden des der SYNC Nachricht werden alle Transmit-PDOs

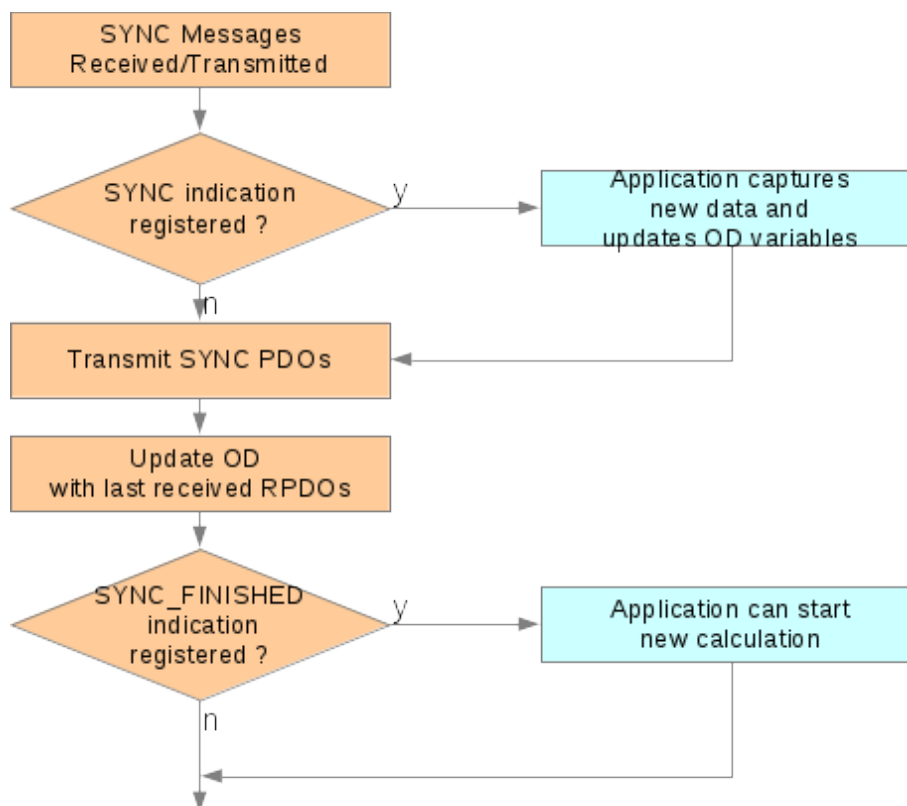


Abbildung 6: PDO Sync

mit den Daten aus dem Objektverzeichnis versendet, und alle beim letzten SYNC empfangenen Empfangs-PDOs in das Objektverzeichnis übernommen. Über die registrierten Indikation Funktionen können dabei die Daten im Objektverzeichnis aktualisiert bzw. abgeholt werden.

### 6.4.6 Multiplexed PDOs (MPDOs)

Wenn die Nutzung der Standard-PDOs mit ihrem festen Mapping nicht ausreicht, kann eine Sonderform der PDOs genutzt werden. Diese MPDOs übertragen nicht nur die Nutzdaten, sondern auch die zugehörigen Index- und Subindex Informationen. Somit benötigen sie kein festes Mapping, sondern können für beliebige Objekte genutzt werden, die per PDO übertragen werden dürfen. Im Gegensatz zu Standard-PDOs kann aber immer nur ein Objekt übertragen werden.

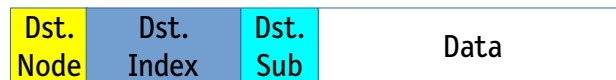
Mit der Funktion `register_MPDO()` kann eine Indikation Funktion angemeldet werden, die beim Eintreffen eines MPDOs aufgerufen wird.

Das Senden von MPDOs erfolgt mit der Funktion `coMPdoReq()`.

Achtung! MPDOs können nur asynchron übertragen werden und müssen daher den Transmission Type 254/255 besitzen.

#### 6.4.6.1 MPDO Destination Address Mode (DAM)

Im Destination Address Mode werden im PDO die Empfänger (Consumer) Informationen mit übertragen, wo die Daten gespeichert werden sollen:



##### 6.4.6.1.1 MPDO DAM Producer

Einträge im Objektverzeichnis

Index	SubIndex	Beschreibung	Wert
18xx <sub>h</sub>		PDO Kommunikationsparameter	
1Axx <sub>h</sub>	0	Anzahl Mapping Einträge	255
1Axx <sub>h</sub>	1	Mapping Eintrag	Appl spezifisch

##### 6.4.6.1.2 MPDO DAM Consumer

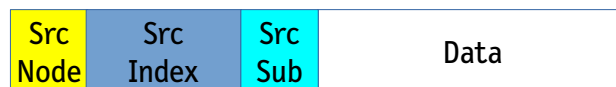
Einträge im Objektverzeichnis

Index	SubIndex	Beschreibung	Wert
14xx <sub>h</sub>		PDO Kommunikationsparameter	
16xx <sub>h</sub>	0	Anzahl Mapping Einträge	255

Die empfangenen Daten werden in den übertragenen Index/Subindex auf dem Consumer gespeichert.

#### 6.4.6.2 MPDO Source Address Mode (SAM)

Im Source Address Mode werden im PDO die Sender (Producer) Informationen mit übertragen, von welchem Knoten die Daten gesendet wurden:



##### 6.4.6.2.1 MPDO SAM Producer

Der SAM Producer nutzt eine Objekt-Scanner Liste, in der alle Objekte hinterlegt sind, die mit dem MPDO versendet werden dürfen. Pro Gerät ist nur 1 MPDO im SAM Producer Mode erlaubt.

Einträge im Objektverzeichnis

Index	SubIndex	Beschreibung	Wert
18xx <sub>h</sub>		PDO Kommunikationsparameter	
18xx <sub>h</sub>	2	Transmissin Type	254/255
1Axx <sub>h</sub>	0	Anzahl Mapping Einträge	254
1FA0 <sub>h</sub> ..1FCF <sub>h</sub>	0-254	Scanner Liste	

Die Scanner Liste hat das folgende Format:

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

#### 6.4.6.2 MPDO SAM Consumer

Einträge im Objektverzeichnis

Index	SubIndex	Beschreibung	Wert
14XX <sub>h</sub>		PDO Kommunikationsparameter	
16XX <sub>h</sub>	0	Anzahl Mapping Einträge	254
1FDO <sub>h</sub> ..1FFF <sub>h</sub>	0-254	Dispatcher Liste	

Die Dispatcherliste liefert eine Cross-Referenz zwischen den Remote Objekt und dem Objekt im lokalen Objektverzeichnis. Die empfangenen Daten werden dann anhand der Dispatcher Liste auf dem Consumer gespeichert.

DispatcherListe

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

Die Blocksize erlaubt die Beschreibung von gleichartigen Subindexen.

## 6.5 Emergency

### 6.5.1 Emergency Producer

Das Senden von Emergency Nachrichten kann durch die Applikation angewiesen, aber auch automatisch bei bestimmten Fehlersituationen (CAN Bus-Off, falsche PDO Länge, ...) erfolgen. In diesem Fall können die 5 Byte applikationsspezifischen Fehlercodes über die registrierte Event Funktion (siehe `registerEvent_EMCY`) durch den Anwender gesetzt oder auch das Senden verhindert werden.

### 6.5.2 Emergency Consumer

Emergency Consumer werden über ihre CAN-ID im Objektverzeichnis im Objekt 0x1028 eingetragen. Alle dort konfigurierten CAN-IDs werden beim Eintreffen als Emergency Nachricht interpretiert und über die registrierte Event Funktion (`registerEvent_EMCY\_CONSUMER`) der Applikation als empfangene Emergency-Nachricht zur Verfügung gestellt.



## 6.6 NMT

Statuswechsel werden standardmäßig vom NMT Master gesendet und müssen von den NMT Slaves umgesetzt werden. Einzige Ausnahme davon ist der Übergang nach OPERATIONAL. Dieser wird nur durchgeführt, wenn die registrierte Event Funktion (siehe registerEvent\_NMT) ohne Fehler zurückkehrt.

Selbständige Statuswechsel darf der Knoten nur in Fehlerfällen (Heartbeat Ausfall, CAN Bus-Off) durchführen, wenn er sich im Zustand OPERATIONAL befindet. Maßgebend dafür sind die Einstellungen im Objekt 0x1029, welche durch den CANopen Protokoll Stack berücksichtigt werden.

### 6.6.1 NMT Slave

NMT Slave Geräte müssen die vom NMT Master gesendeten Nachrichten umsetzen. Empfangene NMT Kommandos können über die Event-Funktion registerEvent\_NMT der Applikation mitgeteilt werden.

### 6.6.2 NMT Master

Der NMT Master kann mit der Funktion coNmtStateReq() die NMT Zustände aller Knoten im Netzwerk umschalten. Dies kann individuell für jeden Knoten, oder netzwerkweit erfolgen. In diesem Fall legt ein zusätzlicher Parameter fest, ob das Kommando auch für den eigenen Masterknoten gelten soll.

### 6.6.3 Default Error Behaviour

Das Verhalten im Fehlerfall (Heartbeat Consumer Event oder CAN Bus Off) wird über das Objekt 0x1029 festgelegt. Wenn das Objekt nicht existiert, wechselt der Knoten defaultmäßig in den Zustand PRE-OPERATIONAL. Bei aktiviertem Emergency-Producer wird automatisch eine Emergency Nachricht versendet. Wenn die Emergency Funktion (registerEvent\_EMCY) registriert ist, kann hier die 5 additional Bytes vor dem Versenden modifiziert werden.

## 6.7 SYNC

Das Senden der SYNC-Nachricht startet, sobald im Objekt 0x1005 das Sync-Producerbit gesetzt und eine Zeit ungleich 0 im Objekt 0x1006 eingetragen ist.

Für das SYNC sind 2 Indikation Funktionen vorgesehen (siehe registerEvent\_SYNC und registerEvent\_SYNC\_FINISHED):

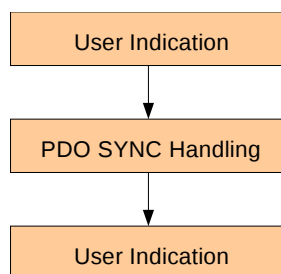


Abbildung 7: SYNC Handling

## 6.8 Heartbeat

### 6.8.1 Heartbeat Producer

Das Eintragen einer neuen Heartbeatzeit wird sofort übernommen. Zeitgleich wird die erste Heartbeat Nachricht versendet, wenn der eingetragene Wert ungleich null ist.

Hintergrund: Eine lange Heartbeatzeit könnte kurz vor dem Ablaufen sein. Selbst wenn nun eine kürzere Zeit eingetragen wird, könnte damit das Heartbeat-Intervall bei den Consumern überschritten sein.

### 6.8.2 Heartbeat Consumer

Das Einrichten bzw. Löschen von Heartbeat Consumern kann entweder über die Funktion [coHbConsumerSet\(\)](#) oder über das Eintragen in die Objekte 0x1016:1..n erfolgen.

Wenn die Funktion [coHbConsumerSet\(\)](#) genutzt wird, wird der Heartbeat Consumer automatisch im Objektverzeichnis auf dem Index 0x1016 eingetragen, wenn noch ein freier Eintrag verfügbar ist. Ansonsten kehrt die Funktion mit einem Fehler zurück.

Bootup Nachrichten werden von allen Knoten empfangen, auch wenn sie nicht in der Heartbeat Consumer Liste eingetragen sind.

Bei Änderungen des Überwachungsstatus wird die registrierte Event Funktion (siehe [registerEvent\\_ERRCTRL](#)) aufgerufen. Diese können sein:

CO_ERRCTRL_BOOTUP	Bootup Nachricht empfangen
CO_ERRCTRL_NEW_STATE	NMT Status geändert
CO_ERRCTRL_HB_STARTED	Heartbeat Überwachung startet
CO_ERRCTRL_HB_FAILED	Heartbeat ausgefallen
CO_ERRCTRL_GUARD_FAILED	Guarding vom Master ausgefallen
CO_ERRCTRL_MGUARD_TOGGLE	Toggle Fehler vom Slave
CO_ERRCTRL_MGUARD_FAILED	Guarding vom Slave ausgefallen
CO_ERRCTRL_BOOTUP_FAILURE	Fehler beim Senden der Bootup

### 6.8.3 Life Guarding

Das Life Guarding wird automatisch aktiviert, wenn die Einträge in den Objekten 0x100c und 0x100d ungleich 0 sind und das erste Guarding vom Master empfangen wurde. Nach Ablauf der in den Objekten eingestellten Zeit bzw. dem Lifetimefaktor wird das Standard Fehlerverhalten (siehe Kapitel 6.6.3 Default Error Behaviour) ausgeführt, und die registrierte Indikation Funktion (siehe [registerEvent\\_ERRCTRL](#)) aufgerufen.

## 6.9 Time

Der Time Dienst kann als Producer oder Consumer genutzt werden. Bei der Initialisierung ist festzulegen, welche Modi (Producer und/oder Consumer) verfügbar sein sollen.

Für das Versenden des Time\_Dienstes steht die Funktion [coTimeWriteReq\(\)](#) zur Verfügung. Eintreffende Time-Nachrichten werden über die angemeldete Indikation Funktion (siehe registerEvent\_[TIME](#)) angezeigt.

### 6.10 LED

Für die Signalisierung entsprechend CiA 303 können 2 LEDs (Status und Error-LED) angesteuert werden. Entsprechend dem aktuellen NMT und Fehlerstatus werden diese über die registrierte Event Funktion (siehe registerEvent\_LED) ein- bzw. ausgeschaltet.

### 6.11 LSS Slave

Für den LSS Dienst existiert eine eigene, vom NMT Zustand unabhängige Statemachine:


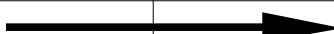
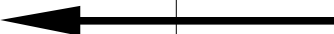
Status	Bedeutung
LSS Waiting	Normalzustand
LSS Configuration	Konfigurationszustand, Node-Id und Bitrate kann eingestellt werden

Die Umschaltung der beiden Stati erfolgt über den LSS Master. Diese werden über die mit [coEventRegister\\_Lss\(\)](#) übergebene Funktion angezeigt.

Für den LSS Slave Dienst werden intern 3 Node-Id Werte verwaltet:

Persistent Node-Id	Power-On Value, wird durch die Applikation bereitgestellt
Pending Node-Id	Temporäre Node-Id
Active Node-Id	Aktive Node-Id des Gerätes

NMT Status Wechsel und interne Events bewirken ein umkopieren der Node-Ids:

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

Die [Active Node-Id](#) wird beim Reset Communication von der [Pending Node-Id](#) übernommen. Das Reset Communication muss dabei vom NMT-Master kommandiert werden.

Startet der Knoten mit einer [Persistent Node-Id](#) = 255 und bekommt mit „LSS Set Node Id“ eine

Knotennummer zugewiesen, erfolgt ein automatisches Reset Communication beim Übergang in den LSS Zustand *Waiting*.

Die **Persistent Node Id** muss von der Applikation als Standard Node-Id bereitgestellt werden. Wenn die **Persistent Node-Id** nichtflüchtig speicherbar und damit zur Laufzeit modifiziert werden kann, muss die Standard Node-Id über eine Funktion bereitgestellt werden. Anderenfalls wird die Node-Id bei einem Reset Application nicht korrekt übernommen.

Kommandos vom LSS Master werden ebenfalls über die mit `coEventRegister_Lss()` übergebene Funktion der Applikation übergeben. Bei einem „LSS Store Kommando“ muss die übergebene Node-Id (=> **Persistent Node-Id**) im nichtflüchtigen Speicher gesichert und beim Aufruf der Node-Id Funktion bereitgestellt werden. Alternativ kann die Node-Id auch als Konstante festgelegt werden. In diesem Fall muss das „LSS Store Kommando“ mit einem Fehlerrückgabewert abgewiesen werden.

## 6.12 Configuration Manager

Der Configuration Manager kann nur von NMT-Master Geräten genutzt werden und dient zur Konfiguration von NMT-Slaves mit Hilfe von vorkonfigurierter DCF-Dateien. Die DCF-Dateien können dabei im ASCII- als auch im Concise-DCF Format vorliegen.

Für die Übertragung der Daten zu den Slaves muss die Konfiguration als Concise-DCF im Objekt 0x1F22 vorliegen. Die Consice-DCF Daten können direkt in diesem Objekt bereitgestellt, oder über die Konvertierungsfunktion `co_cfgConvToConsive()` aus einer Standard-DCF Datei konvertiert werden. Der Funktion sind entsprechende Puffer zu übergeben, so dass auch eine partielle Umwandlung der DCF Daten erfolgen kann.

Die Konfiguration wird für jeden Slave einzeln über die Funktion `co_cfgStart()` eingeleitet. Wenn die Objekte 0x1F26 und 0x1F27 (expected configuration date/time) verfügbar sind, prüft die Funktion die Einträge in dem zugehörigen Slave (Objekt 0x1020), ob im Slave schon die aktuelle Konfiguration enthalten ist. Wenn dies nicht der Fall ist oder die Objekte nicht verfügbar sind, erfolgt die Übertragung der Konfigurationsdaten. Das Ende der (erfolgreichen oder fehlerhaften) Konfiguration wird über die angemeldete Indikation Funktion (siehe `registerEvent_CONFIG`) angezeigt.

Die Konfiguration der Slaves erfolgt über SDO Transfers. Daher muss mindestens ein SDO Client eingerichtet sein. Die parallele Konfiguration von mehreren Slaves mit mehreren SDO Clients ist möglich.

Achtung: Während der Konfiguration können diese SDOs nicht von der Applikation genutzt werden!

## 6.13 Flying Master

Für die Nutzung der Flying Master Funktionalität muss das Objekt 0x1f80 vorhanden und das Flying Master Flag zwingend gesetzt sein. Beim Systemstart beginnt das Gerät als Slave und startet die Master Aushandlung automatisch. Das Ergebnis der Masteraushandlung wird über die mit `coRegister_FLYMA()` übergebene Funktion signalisiert. Wenn der Knoten auf Grund seiner Priorität als Slave arbeitet muss, muss die Heartbeatüberwachung des Master von der Applikation eingerichtet werden. Beim Ausfall des aktiven Masters wird dann automatisch eine neue Masteraushandlung gestartet.

### 6.14 Kommunikations-Status Auswertung

Kommunikationsstatus-Übergänge können durch die Hardware getriggert (Bus-Off, Error Passiv, Overflow, Nachrichteneingang, Sende-Interrupt)), oder durch einen Timer ausgelöst werden (Return vom Bus-OFF). Diese werden über die registrierte Event Funktion (siehe `coRegisterEvent_COMM_EVENT`) gemeldet.

Die Kommunikation Status Auswertung umfasst:

- Auswertung des CAN Controller Status
- Status der Sende- und Empfangs-Queue

Welcher Statusübergang einen Wechsel des Kommunikationszustands bewirkt, zeigt folgende Tabelle:

Statuswechsel/ Event	Eingenommener Status (Kommunikationsstatus)	Beschreibung
Bus-OFF	Bus-OFF	CAN Controller ist im Bus-Off, keine Kommunikation möglich
Bus-OFF Recovery	Bus-OFF	CAN Controller versucht aus Bus-Off Status in aktiven Zustand zu wechseln
Return vom Bus-OFF	Bus-On	CAN Controller ist wieder kommunikationsbereit und konnte wenigstens eine Nachricht senden oder empfangen
Error Passive	Bus-on, CAN passiv	CAN Controller im Error Passive Zustand
Error Active	Bus on	CAN Controller im Error Active Zustand
CAN Controller overrun	-	Nachrichten sind im CAN Controller überschrieben worden, weil sie nicht schnell genug ausgelesen wurden. Wird bei jedem Nachrichtenverlust aufgerufen
REC-Queue full	-	Empfangsqueue ist voll
REC-Queue overflow	-	Nachrichten sind verloren gegangen. Wird bei jedem Nachrichtenverlust aufgerufen
TR-Queue full	Bus-Off/On, Tr-Queue voll	Sende Queue ist voll, aktuelle Daten werden gespeichert, neue Daten können nicht mehr gespeichert werden
TR-Queue overflow	Bus-Off/On, Tr-Queue overflow	Sende Queue ist voll, auch aktuelle Daten können nicht mehr gespeichert werden.
TR-Queue empty	Bus-On, Tr-Queue bereit	Sende Queue ist mindestens zur Hälfte geleert.

### 6.15 Sleep Mode für CiA 447 und CiA 454

Der Sleep Mode entsprechend CiA 454 kann als Master oder Slave genutzt werden. Die aktuelle Sleep-Mode Phase kann über die mit `coEventRegister_SLEEP()` angemeldete Funktion ausgewertet bzw. eingenommen werden.

Der Sleep Mode wird vom NMT Master kommandiert und besteht aus mehreren Phasen:

NMT Master Funktion	Phase	Slave
coSleepModeCheck()	Sleep Check	prüft, ob Sleep Mode eingenommen werden kann. Wenn nicht, erfolgt eine Rückmeldung an NMT Master
coSleepModeStart()	Sleep Prepare	Sleepmode vorbereiten, Applikation herunterfahren, Kommunikation ist noch möglich, Sleep Timer 1 starten
(timergesteuert)	Sleep Silent	Kein Senden auf CAN, Empfang aber noch möglich, Sleep Timer 2 starten
(timergesteuert)	Sleep	Schlafmode

Die vom Master gestartete Phase „Sleep Prepare“ wird automatisch über einen Timer in die weiteren Phasen überführt. Die Phasen gelten sowohl für die NMT Slaves als auch für den NMT Master selber, und werden mit der angemeldeten Funktion signalisiert. Die Applikation muss in der „Sleep“ Phase die Indikation Funktion nicht mehr verlassen.

Das Aufwachen aus dem Schlafmode erfolgt, sobald CAN Traffic auf dem Bus erkannt wird. Die Aufgabe der Applikation ist, alle Applikationsdaten auf den Stand zu bringen, der vor dem Sleep Mode vorhanden war und die [coSleepAwake\(\)](#) Funktion einmalig aufzurufen. Dies führt zu einem Reset Kommunikation und dem Versenden der wakeup-Nachricht.

Mit der Funktion [coSleepModeActive\(\)](#) kann geprüft werden, ob einer Phase der Sleepmodes aktiv sind.

## 6.16 Startup Manager

Für die Nutzung des Startup Managers müssen folgende Voraussetzungen erfüllt sein:

- Objekt 0x1f80 NMT Master muss vorhanden und entsprechend gesetzt sein
- Für jeden Slave müssen die Eigenschaften im Objekt 0x1f81 definiert sein (Subindex entspricht der Slave Node-Id)
- Die Boot Time (Objekt 0x1f89) muss auf die maximale Bootzeit gesetzt werden
- Für jeden Slave muss ein Client SDO bereitgestellt werden (Subindex entspricht der Slave Node-Id)

Mit der Funktion [coManagerStart\(\)](#) wird der Bootup Prozess entsprechend des CiA 302-2 gestartet. Alle dafür notwendigen Informationen werden aus den Objekten 0x1f80..0x1f89 entnommen. Zugehörige Events wie Start, Stop, Fehler, User-Interaktion werden über die mit [coEventManagerManagerBootup\(\)](#) konfigurierte Funktion der Applikation mitgeteilt. Check- und Update der Slave Software und Update der Konfiguration müssen durch die Applikation ausgeführt werden. Die Fortsetzung des Bootup Prozesses erfolgt mit den entsprechenden Funktionsaufrufen:

Event	Aufgabe Applikation	Fortsetzung mit
CO_MANAGER_EVENT_UPDATE_SW	Check und Update Slave Firmware	coManagerContinueSwUpdate
CO_MANAGER_EVENT_UPDATE_CONFIG	Update Slave Konfiguration	coManagerContinueConfigUpdate
CO_MANAGER_EVENT_RDY_OPERATIONAL	OPERATIONAL für Knoten	coManagerContinueOperational

## 7 Timer Handling

Das Timer Handling basiert auf einem zyklischen Timer, dessen Timerintervall individuell für jede Applikation festgelegt werden kann (auch externer Timer möglich). Ein Timerintervall wird als Timertick bezeichnet. Darauf werden alle zeitabhängigen Vorgänge abgebildet, so dass alle Timervorgänge in Timerticks berechnet werden können.

Ein neues Timerereignis wird mit der Funktion `coTimerStart()` in die verkettete Timer-Liste einsortiert, so dass alle zeitlichen Vorgänge entsprechend ihrer Ablaufzeit hintereinander stehen. Somit muss nach Ablauf eines Timerticks nur der erste Timervorgang geprüft werden, da alle weiteren Timer noch nicht abgelaufen sein können.

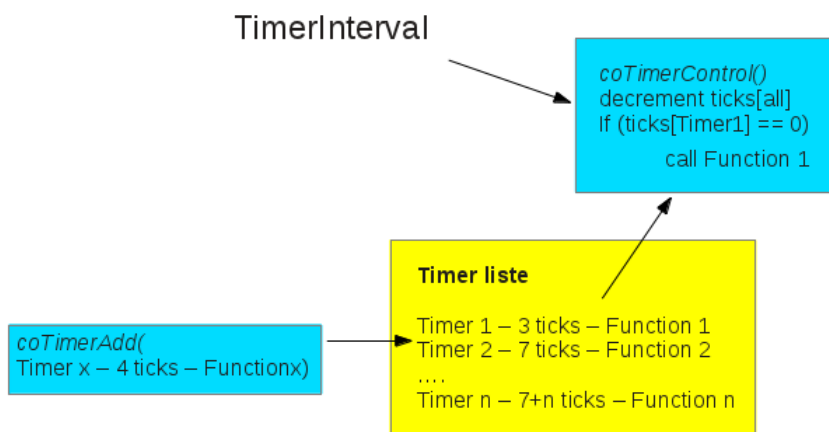


Abbildung 8: Timer Handling

Die notwendigen Timerstrukturen müssen von der aufrufenden Funktion bereit gestellt werden. Damit ergeben sich auch keine Einschränkungen hinsichtlich der Anzahl der Timer.

Beim Ablauf eines Timers wird zuerst der Timer aus der Liste entfernt, und dann die vorgesehene Funktion aufgerufen, die bei der Initialisierung übergeben wurde.

Da nicht alle Zeiten ein Vielfaches der Timerticks sein werden, wird die angegebene Zeit gerundet. In welche Richtung (auf- oder abrunden) dies geschieht, kann bei der Funktion `coTimerStart()` als Parameter übergeben werden.

## 8 Treiber

Der Treiber besteht aus einem CPU- und einem CAN-Teil.

### CPU-Treiber

Der CPU Treiber hat die Aufgabe, einen konstanten Timer-Takt zur Verfügung zu stellen. Dieser kann entweder mit einem eigenen Hardwareinterrupt erzeugt werden, oder von einem anderen Applikations-

Timer abgeleitet werden.

## CAN-Treiber

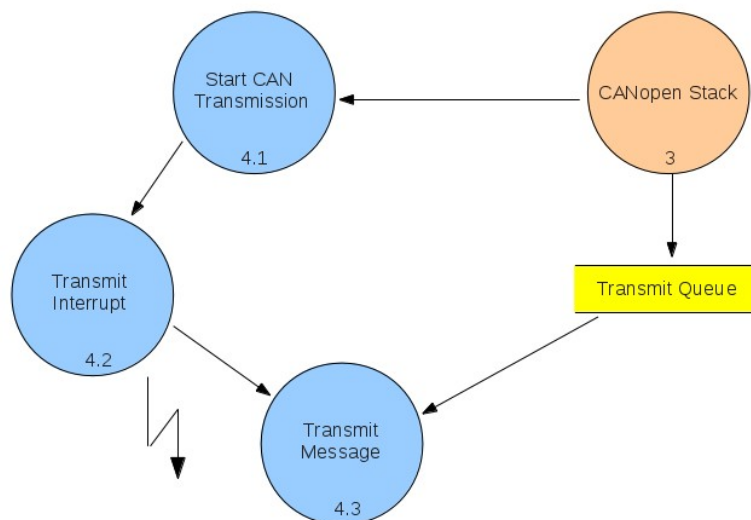
Aufgabe des CAN Treibers ist das Senden und Empfangen von CAN Nachrichten, sowie das Bereitstellen des aktuellen CAN Status. Das Pufferhandling erfolgt direkt im CANopen Protokoll Stack.

### 8.1 CAN Transmit

Sendenachrichten werden vom Stack zuerst in den Sendepuffer geschrieben. Anschließend wird das Senden mit der Funktion `codrvCanStartTransmission()` angestoßen.

Das Senden der Nachrichten erfolgt interruptgesteuert. Daher muss in der Funktion `codrvCanStartTransmission()` nur der Sendeinterrupt ausgelöst werden.

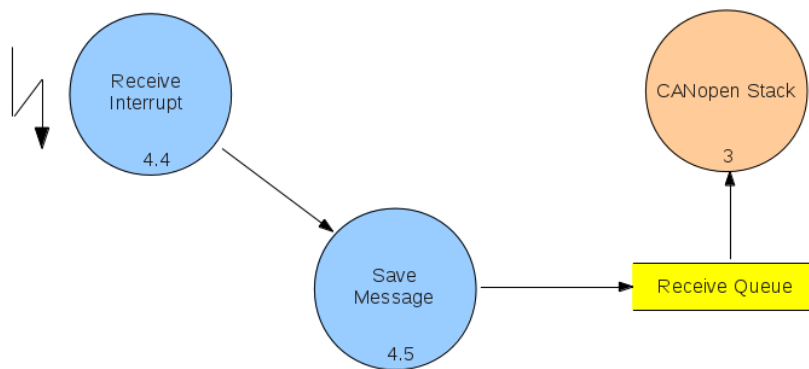
Im Transmit-Interrupt wird mit der Funktion `codrvCanTransmit()` die nächste Nachricht aus dem Sendepuffer geholt, in den CAN Controller geschrieben und versendet. Dies wird solange wiederholt, bis alle Nachrichten aus dem Sendepuffer versendet sind.



### 8.2 CAN Receive

Der Empfang von CAN Nachrichten erfolgt interruptgesteuert. Dabei wird die empfangene Nachricht direkt in die Receive-Queue geschrieben, und kann anschließend vom CANopen Stack gelesen und verarbeitet werden.





## 9 Einbindung mit Betriebssystemen

Für die Nutzung des Stacks mit Betriebssystemen stehen 2 Möglichkeiten zur Verfügung:

1. Implementierung des Stacks in einer Task und zyklischer Aufruf der zentrale Bearbeitungsfunktion
2. Aufteilung in verschiedene Task

Dafür ist eine entsprechende Intertask-Kommunikation einzurichten

### 9.1 Aufteilung in mehrere Tasks

Durch die Aufteilung in verschiedene Tasks ist kein Polling der zentrale Bearbeitungsfunktion notwendig. Sie bleibt aber aus Kompatibilitätsgründen weiterhin als zentrale Funktion erhalten und entscheidet intern, welche Funktionalität abzuarbeiten ist. Sie ist bei folgenden Ereignissen aufzurufen:

- CAN Sendeinterrupt
- CAN Empfangsinterrupt
- CAN Statusinterrupt
- Timerinterrupt

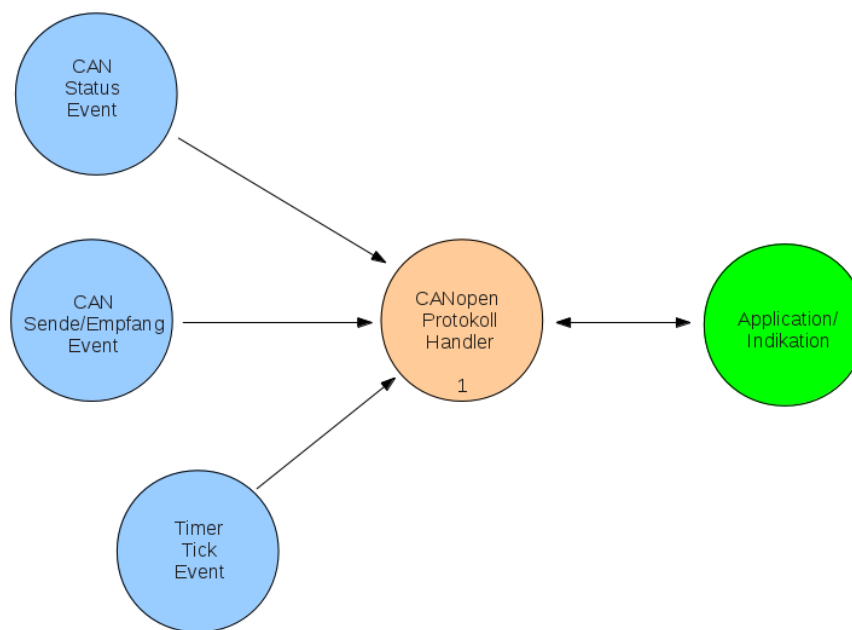


Abbildung 9: Prozess Signal Handling

Welche Interprozess-Aktivierung verwendet wird, ist vom verwendeten Betriebssystem abhängig und über die Makros

Makro	Verwendung vor/in	Bedeutung
CO_OS_SIGNAL_WAIT()	coCommTask()	wartet auf Signal
CO_OS_SIGNAL_TIMER()	Timer handler	signalisiert Timer Tick
CO_OS_SIGNAL_CAN_STATE()	CAN status interrupt	signalisiert geänderten CAN Status
CO_OS_SIGNAL_CAN_RECEIVE()	CAN Empfangs Interrupt	signalisiert neue CAN Nachricht
CO_OS_SIGNAL_CAN_TRANSMIT()	CAN Sende Interrupt	signalisiert versendete CAN Nachricht

festzulegen.

## 9.2 Objektverzeichniszugriff

Bei der Aufteilung in verschiedene Tasks ist auch der Schutz des Objektverzeichnisses zu gewährleisten. Innerhalb des Stacks werden dafür die Makros

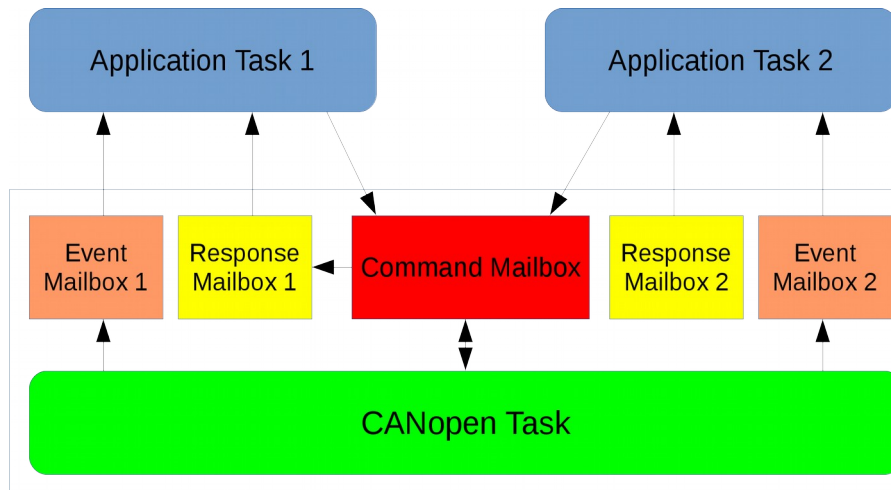
CO_OS_LOCK_OD	Lock des Objektverzeichnis
CO_OS_UNLOCK_OD	UnLock des Objektverzeichnis

genutzt. Diese sind auch in der Applikation entsprechend anzuwenden.

Innerhalb des Stacks erfolgt das Lock bzw. Unlock direkt vor bzw. nach dem Zugriff auf die entsprechenden Objekte.

### 9.3 Mailbox-API

Die Mailbox-API bietet eine alternative API für die Funktionen und Benachrichtigungen des CANopen-Stacks an. Der CANopen-Stack läuft dabei in einem separaten Thread/Task. Von einer beliebigen Anzahl von Applikationsthreads<sup>1</sup> können Kommandos an den CANopen-Thread über Messagequeues gesendet werden. Das Senden von Kommandos entspricht damit dem Funktionsaufruf einer herkömmlichen API-Funktion.



Der CANopen-Thread sendet zu jedem Kommando eine Response mit dem Rückgabewert des Kommandos über eine Response-Queue. Zusätzlich können Benachrichtigungen über Events über eine Eventqueue bereitgestellt werden. Bei der Konfiguration der Eventqueue kann festgelegt werden, über welche Events (z.B. PDO Empfang, NMT Statusänderung, usw.) der jeweilige Applikationsthread informiert werden soll. Dieses Eventhandling ersetzt die Indikationsfunktionen der herkömmlichen Funktions-API.

Aktuell ist die Mailbox-API für die Betriebssysteme QNX, Linux und RTX64 implementiert, es kann jedoch auf jedes Betriebssystem portiert werden, welches Queues unterstützt.

#### 9.3.1 Einrichtung eines Applikationsthreads

Jeder Applikationsthread besteht aus einem Initialisierungsteil und einem zyklischen Hauptteil. Im Initialisierungsteil muss sich mit der Kommandoqueue des CANopen-Threads verbunden werden und optional können Thread-spezifische Response- und Event-Queues angelegt werden, wie das nachfolgende Beispiel zeigt:

```
/* 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);
}
```

<sup>1</sup> Nachfolgend wird der Begriff Thread verwendet. Ob es tatsächlich Threads oder Tasks sind hängt vom verwendeten Betriebssystem ab.

```

}

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

```

Nach dem Einrichten der Mailboxen muss für die Event-Mailbox definiert werden, über welche Events der Applikationsthread benachrichtigt werden soll. Dies zeigt das nachfolgende Beispiel:

```

/* 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); };

```

Zur Registrierung für weitere Events wird auf das Referenzhandbuch und das Beispiel verwiesen.

### 9.3.2 Senden von Kommandos

Für alle grundlegende CANopen-Funktionen und wichtige CANopen-Master-Funktionen<sup>2</sup> sind entsprechenden Mailbox-Kommandos angelegt. Zum Senden eines Kommandos sind die entsprechenden Structs zu füllen, deren Member den Argumenten der dazugehörigen CANopen-Funktion entsprechen. Nachfolgend ein Beispiel zur Veranschaulichung:

```

/*-----*
 * 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);

```

Der Rückgabewert von requestCommand() ist dabei eine selbstständig hochlaufende Nummer, welche bei der Response vom CANopen-Thread zurücksendet wird und somit eine Zuordnung von Kommando und Response(Rückgabewert der Funktion) ermöglicht.

---

*/\* wait for new messages for 1ms \*/*

<sup>2</sup> Weitere CANopen-Funktionen können bei Bedarf als Kommando implementieren werden.

```
Mbx_WaitForResponseMbx(mqResp, &response, 1);
```

Die Response beinhaltet das Kommando (z.B. MBX\_CMD\_NMT\_REQ), die fortlaufende Kommandonummer sowie den Rückgabewert der darunterliegenden CANopen-Funktion vom Typ RET\_T.

Folgende CANopen-Funktionen werden aktuell durch das Mailbox-API unterstützt:

CANopen-Funktion	Kommando
<i>coEmcyWriteReq()</i>	MBX_CMD_EMCY_REQ
<i>coPdoReqNr()</i>	MBX_CMD_PDO_REQ
<i>coNmtStateReq()</i>	MBX_CMD_NMT_REQ
<i>coSdoRead()</i>	MBX_CMD_SDO_RD_REQ
<i>coSdoWrite()</i>	MBX_CMD_SDO_WR_REQ
<i>coOdSetCobId()</i>	MBX_CMD_SET_COBID
<i>coOdGetObj_xx()</i>	MBX_CMD_GET_OBJ
<i>coOdPutObj_xx()</i>	MBX_CMD_PUT_OBJ
7 coLss... Funktionen	MBX_CMD_LSS_MASTER_REQ

Zur Beschreibung der Funktionen(Kommandos) und deren Rückgabewerte(Responses) wird auf das Referenzhandbuch verwiesen.

### 9.3.3 Empfang von Events

Nachdem der Empfang von CANopen-Events durch den Applikationsthread registriert wurden, können CANopen-Events über *Mbx\_WaitForEventMbx()* empfangen werden. Die möglichen Events entsprechend den Indikationen und die Member der Event-Struktur entsprechend den Argumenten der registrierbaren Indication-Funktionen.

```
/* 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 event */
    }
}
```

```

    default:
        break;
}

```

## 10 Multi-Line Handling

Die Nutzung des Multi-Line Stacks erfolgt analog dem der Single-Line Version. Damit stehen auch hier alle Funktionen der Single-Line Version zur Verfügung. Die Daten für jede Linie werden getrennt verwaltet, so dass die Linien unabhängig voneinander betrieben werden können. Die Erstellung des Objektverzeichnisses erfolgt mit dem CANopen DeviceDesigner in einem gemeinsamen Projekt, aber für jede Linie getrennt.

Jede Funktion erhält als ersten Parameter die Linien Information als UNSIGNED8 Wert. Die erste Linie beginnt mit dem Wert 0. Dies gilt nicht nur für die Stack-Funktionen, sondern auch für alle Indikation Funktionen.

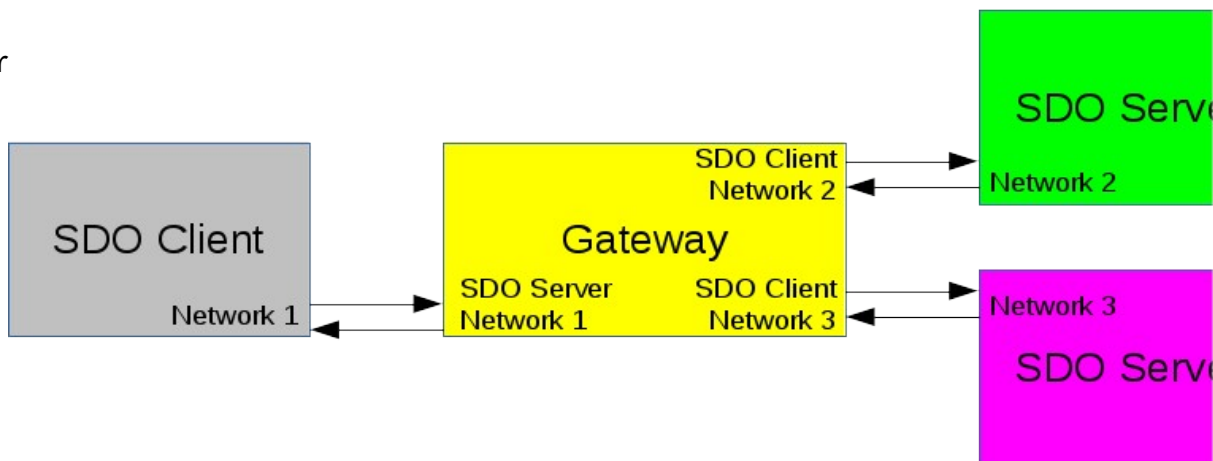
Die Beispiele für Multi-Line sind unter `example/ml_xxx` zu finden.

## 11 Multi-Level Networking – Gateway Funktionalität

Für das Networking sind die Routen im Objekt `0x1F2c` zu hinterlegen. Hier wird festgelegt, welche Netzwerke über welches CAN-Interface erreicht werden kann.

### 11.1 SDO Networking

Der



SDO Client initiiert analog zu den Standard SDO Kommunikation die Verbindung zum Gateway, in dem er neben dem Index und Subindex auch das gewünschte Netzwerk und die Knotennummer übergibt.

Das Gateway leitet nun alle Anfragen des Clients an den gewünschten Server. Dafür muss das Gateway die Daten als SDO Server annehmen, und eine neue Kommunikation zum gewünschten Server in dem anderen Netzwerk als SDO Client aufbauen. Den zu nutzenden SDO Client kann dem Stack über die mit `register_GW_CLIENT()` registrierte Funktion vorgegeben werden. Wenn keine Funktion festgelegt wurde, wird immer SDO Client 1 genutzt. Ist das SDO Client 1 nicht verfügbar, kann das Gateway keine Kommunikation aufbauen. Die COB-IDs für das Client SDO werden dann automatisch für den Zugriff auf den SDO Server gesetzt. Die SDO COB-Ids werden nach der Beendigung des Transfers nicht rückgesetzt.

## 11.2 EMCY Networking

Für das Emergency Networking ist die Emergency Routing Liste (Objekt 0x1f2f) auszufüllen. Sie enthält bitcodiert die Netzwerknummer, zu denen die EMCY Nachricht weitergeleitet wird. Die Subindexe korrelieren mit Emergency Consumerliste (Objekt 0x1028) und werden parallel ausgewertet.

## 11.3 PDO Forwarding

Das PDO Forwarding erfolgt automatisch für alle Objekte im Bereich 0xB000 bis 0xBfff, die in ein Empfangs- oder ein Sende-PDO gemappt sind. Dabei ist im CANopen DeviceDesigner zu beachten, dass die Objekte nur in einer Linie angelegt und als „shared in all lines“ deklariert sind.

Generell kann einem Empfangs-PDO nur ein Sende-PDO pro Linie zugeordnet werden, da die Forwarding-Liste bei dem jeweiligen Empfangs-PDO hinterlegt ist. Bei statischen PDOs kann diese Liste während der Laufzeit daher nicht modifiziert werden, auch wenn das Mapping des Sende-PDOs modifiziert wird.

Das Update der Forwarding-Liste erfolgt nach jedem Modifizieren des PDO Mappings.

# 12 Beispiel Implementierung

Um schnell ein CANopen Gerät generieren zu können, stehen mehrere Beispiele zur Verfügung.

Die notwendigen Schritte sind von der konkreten Entwicklungsumgebung abhängig, die prinzipielle Herangehensweise ist aber identisch. Als Grundlagen wird das Beispiel slave1 genutzt. Es kann entweder kopiert oder direkt genutzt werden.

1. Ins Verzeichnis [examples/slave1](#) wechseln
2. Dienste konfigurieren und Objektverzeichnis erstellen
  - *CANopen DeviceDesigner* starten
  - Menü **File->OpenProject** das Projektfile slave1.cddp einlesen
  - Tab **General Settings** die Anzahl der Sende- und Empfangspuffer, und die Anzahl der aufzurufenden Indikation-Funktion festlegen
  - Tab **Object Dictionary** die Dienste und Objekte eintragen
  - Tab **Device Description** die Einträge für das EDS vornehmen
  - Menü **File->Generate Files** die Konfiguration für den Stack und das Objektverzeichnis generieren lassen
  - Menü **File->Save Project** Daten sichern
3. Projekt bzw. Makefile CANopen Sourcen hinzufügen
  - Files unter colib/src (CANopen Stack)
  - Files unter colib/inc (CANopen Stack interne Header)
  - Files unter example/slave1 (Applikation)
  - Files unter codrv/<drivername> (Treiber)
4. Include Pfade setzen
  - examples/slave1
  - colib/inc

- codrv/<drivername>

## 5. Projekt übersetzen

Nun steht ein ausführbares CANopen Projekt zur Verfügung, das entsprechend den Erfordernissen der Applikation angepasst werden kann.

### Files im Beispielprojekt slave1

gen_define.h	Generiertes Files vom <i>CANopen Device Designer</i> , enthält Konfiguration für den Stack
gen_objdict.c	Generiertes Files vom <i>CANopen Device Designer</i> , enthält Objektverzeichnis und Initialisierungsfunktion
main.c	Hauptprogramm
Makefile	Makefile
slave1.cddp	Konfigurationsfile für <i>CANopen DeviceDesigner</i>
slave1.eds	EDS File, generiert durch <i>CANopen DeviceDesigner</i>

## 13 C#-Wrapper

Für Windows sowie Mono unter Linux ist ein C#-Wrapper verfügbar. Die Vorgehensweise ist dabei so, dass das Objektverzeichnis und der eigentliche CANopen-Stack in C zusammen als eine DLL gebaut wird. Die C#-Wrapper-Funktionen nutzen dann die Funktionen aus der applikationspezifischen DLL.

Der C#-Wrapper-Methoden sind alle statisch und in einer Klasse CANopen implementiert. Die Methodennamen entsprechen dabei den Namen der Funktionen in ANSI-C.

Beispiele:

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

Alle Rückgabewerte und Parameter der Methoden entsprechend den C-Pendants und somit kann das Benutzer-Handbuch und das Referenzhandbuch entsprechend genutzt werden.



## 14 Dienste Schritt für Schritt

In diesem Kapitel wird die Einrichtung und Nutzung der einzelnen Dienste beschrieben, insbesondere im Zusammenhang mit dem CANopen DeviceDesigner.

### 14.1 SDO Server Nutzung

*Einrichtung im CANopen DeviceDesigner:*

- Je Server-SDO ein Objekt im Bereich 0x1200 + 0x127F anlegen  
(Hinweis: COB-IDs brauchen nicht gesetzt zu werden, dies muss im Programm erfolgen)
- Bei Blocktransfer die Parameter:
  - Blockgröße für einen Transfer
  - Nutzung von CRC ja/nein
 eintragen

*Einrichtung in der Applikation:*

- Indikation Funktion für Lesen/Schreiben/Test einrichten (`coEventRegister_SDO_SERVER_READ()` / `coEventRegister_SDO_SERVER_WRITE()` / `coEventRegister_SDO_SERVER_CHECK_WRITE()` )
- COB-Id für SDO 1 wird automatisch anhand der Knoten Nummer gesetzt
- ggf. COB-Id für SDO 2..128 setzen (kann aber auch vom Master erfolgen)

*Nutzung in der Applikation:*

- erfolgt asynchron beim Eintreffen von SDO-Anfragen  
der Rückgabewert hat Einfluß auf die Antwort des SDO Transfers

### 14.2 SDO Client Nutzung

*Einrichtung im CANopen DeviceDesigner:*

- Je Client SDO ein Objekt im Bereich 0x1280 + 0x12FF anlegen  
(Hinweis: COB-IDs brauchen nicht gesetzt zu werden, dies muss im Programm erfolgen)
- Bei Blocktransfer die Parameter:
  - Blockgröße für einen Transfer
  - Anzahl der Bytes, ab denen der Blocktransfer genutzt werden soll
  - Nutzung von CRC ja/nein
 eintragen

*Einrichtung in der Applikation:*

- Indikation Funktion für das Ergebnis auf Lesen/Schreiben Aufruf einrichten  
(`coEventRegister_SDO_CLIENT_READ()` / `coEventRegister_SDO_CLIENT_WRITE()` )
- je Client SDO eine COB-Id für Senden und eine COB-Id für den Empfang setzen (kann aber auch vor jeder Nutzung erfolgen)

*Nutzung in der Applikation:*

- COB-Ids entsprechend dem zu kontaktierenden Server setzen
- Transfer starten (`coSdoRead()`, `coSdoWrite()`, `coSdoDomainWrite()` )
- Das Ergebnis des Transfers wird über die eingerichtete Indikation Funktion geliefert
- Bei Nutzung des Domaintransfers (`coSdoDomainWrite()` ) kann zusätzlich eine Indikation Funktion angegeben werden, die nach einer definierten Anzahl von Nachrichten aufgerufen wird, um z.B. den Domainpuffer zu aktualisieren

### 14.3 Heartbeat Consumer

*Einrichtung im CANopen DeviceDesigner:*

- je Heartbeat Consumer einen Subindex im Objekt 0x1016 eintragen
- Überwachungszeit und Knotennummer kann direkt im Objekt eingetragen werden

*Einrichtung in der Applikation:*

- Indikation Funktion für Heartbeat Events einrichten (`coEventRegister_ERRCTRL()` )
- ggf. Überwachungszeiten und Knotennummern neu setzen

*Nutzung in der Applikation:*

- Überwachung startet automatisch beim Eintreffen des ersten Heartbeats
- Jeder Heartbeat Event (Heartbeat gestartet, ausgefallen, geänderter Knotenstatus) wird über die angemeldete Indikation Funktion gemeldet
- Bootup Nachrichten werden von allen Geräten (auch ohne Eintrag in der Heartbeat Consumer Liste) über die angemeldete Indikation Funktion signalisiert

### 14.4 Emergency Producer

*Einrichtung im CANopen DeviceDesigner:*

- Emergency Producer Objekt (0x1014) anlegen
- Error History Objekt (0x1003) mit n-Subindizes einrichten

*Einrichtung in der Applikation:*

- Indikation Funktion für Setzen der herstellerspezifischen Daten einrichten (`coEventRegister_EMCY()` )

*Nutzung in der Applikation:*

- Aufruf durch `coEmcyWriteReq()`
- Eingetragene Indikation Funktion wird automatisch bei PDO Fehlern (zu wenig/zu viel Daten), CAN- oder Heartbeat Fehlern aufgerufen

## 14.5 Emergency Consumer

*Einrichtung im CANopen DeviceDesigner:*

- Emergency Consumer Objekt (0x1028) anlegen
- Eintragen der Emergency Consumer COB-Ids. Der Subindex entspricht dabei den externen Geräteummern.

*Einrichtung in der Applikation:*

- Indikation Funktion für den Empfang der Emergency Nachricht einrichten ([coEventRegister\\_EM CY\\_CONSUMER\(\)](#) )

*Nutzung in der Applikation:*

- Eingetragene Indikation Funktion wird automatisch bei Empfang von konfigurierten Emergency Nachrichten aufgerufen

## 14.6 SYNC Producer/Consumer

*Einrichtung im CANopen DeviceDesigner:*

- SYNC Objekt 0x1005 anlegen
- Producer oder Consumer festlegen (Bit 30 = 1 für Producer)
- für SYNC Producer: SYNC Producer Time Objekt 0x1006 mit Wert in µsec setzen

*Einrichtung in der Applikation:*

- Indikation Funktion für Aktionen beim Eintreffen des SYNCs ([coEventRegister\\_SYNC\(\)](#) ) einrichten
- Indikation Funktion für Aktionen nach der SYNC Behandlung des Stacks ([coEventRegister\\_SYNC\\_FINISHED\(\)](#) ) eintragen

*Nutzung in der Applikation:*

- Eingetragene Indikation Funktionen werden automatisch bei Eintreffen des SYNCs aufgerufen

## 14.7 PDOs

### 14.7.1 Empfangs-PDOs

*Einrichtung im CANopen DeviceDesigner:*

- Objekte, die mit PDO empfangen werden sollen, im Hersteller- (0x2000..0x5FFF) oder im Profibereich (0x6000) anlegen
- PDO Mapping Eintrag von diesen Objekten auf Allowed, RPDO oder TPDO setzen
- PDO Kommunikationsparameter für jedes PDO (Objekte 0x1400.. 0x15ff) anlegen:
  - Transmission Type festlegen – bei synchronen PDOs müssen auch die SYNC Objekte (siehe 14.6) vorhanden sein

- ggf. Event Timer Wert in msec eintragen
- zugehörige PDO Mapping einrichten (Objekte 0x1600..0x17ff)
- Dynamisch/Statisches Mapping festlegen (Reiter Mask: Mapping auf dynamic oder static setzen)

#### *Einrichtung in der Applikation:*

- COB-ID festlegen bzw. modifizieren (PDO1..4 wird automatisch entsprechend dem Pre-defined Connection set eingestellt)
- Indikation Funktion für den Empfang von asynchronen PDOs (`coEventRegister_PDO()`) eintragen
- ggf. Indikation Funktion für Event Timer Empfangsüberwachung einrichten (`coEventRegister_PDO_REC_EVENT()`)
- ggf. Indikation Funktion für EMCY anmelden, um fehlerhaft konfigurierte PDOs zu erkennen
- ggf. Indikation Funktion für SYNC Empfang (`coEventRegister_PDO_SYNC()`) einrichten

#### *Nutzung in der Applikation:*

- Eingetragene Indikation Funktionen werden automatisch nach dem Empfang von PDOs aufgerufen. Dabei sind die neuen Daten schon im Objektverzeichnis hinterlegt.

## 14.7.2 Sende-PDOs

#### *Einrichtung im CANopen DeviceDesigner:*

- Objekte, die mit PDO versendet werden sollen, im Hersteller- (0x2000..0x5FFF) oder im Profilbereich (0x6000) anlegen
- PDO Mapping Eintrag von diesen Objekten auf Allowed, RPDO oder TPDO setzen
- PDO Kommunikationsparameter für jedes PDO (0x1800.. 0x19ff) anlegen:
  - Transmission Type festlegen – bei synchronen PDOs müssen auch die SYNC Objekte (siehe 14.6) vorhanden sein
  - Inhibit Zeit in 100µ festlegen
  - ggf. Event-Timer in msec festlegen
  - ggf. SYNC Start Value festlegen
- zugehörige PDO Mapping einrichten (Objekte 0x1a00..0x1bff)
- Dynamisch/Statisches Mapping festlegen (Reiter Mask: Mapping auf dynamic oder static setzen)

#### *Einrichtung in der Applikation:*

- COB-ID festlegen bzw. modifizieren (PDO1..4 wird automatisch entsprechend dem Pre-defined Connection set eingestellt)

#### *Nutzung in der Applikation:*

- PDO versenden:

- Daten im Objektverzeichnis aktualisieren
- Bei azyklischen oder asynchronen PDOs (Transmission Type 0, 254, 255)
  - `coPdoReqNr()` oder `coPdoReqIndex()` aufrufen
- Bei Synchronen PDOs (Transmission Type 1..240)
  - versenden erfolgt automatisch

## 14.8 Dynamische Objekte

*Einrichtung im CANopen DeviceDesigner:*

- Unter Optional Services → Use Dynamic Objects aktivieren

*Einrichtung in der Applikation:*

- Dynamische Objekte initialisieren mit `coDynOdInit()`
- Dynamisches Objekt hinzufügen mit `coDynOdAddIndex()`
- Dynamischen Subindex hinzufügen mit `coDynOdAddSubIndex()`

*Nutzung in der Applikation:*

- Zugriff erfolgt mit den Standardfunktionen analog zu Objekten, die mit dem CANopen DeviceDesigner angelegt wurden.

## 14.9 Objekt Indikation

*Einrichtung im CANopen DeviceDesigner:*

- Anzahl der gewünschten Objekte mit dem define `CO_EVENT_OBJECT_CHANGED` setzen  
`#define CO_EVENT_OBJECT_CHANGED 5`

*Einrichtung in der Applikation:*

- Indikation anmelden mit `coEventRegister_OBJECT_CHANGED()`

*Nutzung in der Applikation:*

- Die angemeldete Funktion wird immer aufgerufen, wenn das entsprechende Objekt per SDO oder PDO geändert wurde.

## 14.10 Configuration Manager

*Einrichtung im CANopen DeviceDesigner:*

- Objekte 0x1F22 und 0x1F23 (Consive DCF) mit entsprechenden Subindex anlegen
- Objekte 0x1F26 und 0x1F27 (configuration date/time) optional anlegen

- SDO Client(s) 0x1280..0x12ff anlegen

#### *Einrichtung in der Applikation:*

- Indikation Funktion `registerEvent_CONFIG()` anmelden
- Consive-DCF in die Objekte 0x1F22 ablegen
- ggf. DCF Files lesen und in Consice-DCF mit `co_cfgConvToConsive()` wandeln und in Objekte 0x1F22 ablegen

#### *Nutzung in der Applikation:*

- Konfiguration für jeden Slave starten mit `co_cfgStart()`  
Das Ende wird mit der eingestellten Indikation Funktion gemeldet.

## 15 Aufbau der Verzeichnisstruktur

colib/src	CANopen Protokoll Stack Sourcen und interne Header
colib/inc	CANopen Protokoll Stack öffentliche Header (co_canopen.h enthält alle öffentlichen Header)
colib/profile	CANopen Profile Erweiterungen (Source und Header)
colib/csharp_wrapper	C#-Wrapper für CANopen Stack-Funktionen
codrv/	Treiberfiles
examples	Beispielprojekte

## Anhang

### SDO Abortcodes

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