Anwenderhandbuch

CANopen/CANopen FD Master/Slave Protokoll Stack

V 3.12.0

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	
1.4.0	Add SDO Block Transfer	08.07.2013	boe	

Version	Änderungen	Datum	Bearbeiter	Freigabe
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 Indikation/DynOd Applikation	05.07.2015	boe	
2.4.0	MPDO Benutzung hinzugefügt	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	
2.7.0	Anpassung an Lib V2.7	08.05.2017	boe	
2.7.3	Add PDO Update Indikation	16.10.2017	boe	
2.99.0	CAN-FD hinzugefügt	19.12.2017	boe	
3.0.0	Anpassung an V3.0	14.06.2018	boe	
3.2.0	Domain/String-handling	30.01.2019	boe	
3.4.3	Updated Configuration Manager	16.08.2019	hil	
3.5.0	Umstellung zu emotas	07.10.2019	boe	
3.6.0	Neue Version	10.03.2020	boe	
3.7.0	Add CANopen Grundlagen	26.06.2020	boe	
3.7.4	Hinweis auf Anpassung cpu.c/h	03.02.2021	boe	
3.8.0	Neue Version	11.06.2021	boe	
3.9.0	Neue Version	14.10.2021	boe	
3.10.0	Neue Version	04.04.23	boe	
3.11.0	Neue Version	28.08.23	boe	
3.12.0	Add CAN Treiber Anbindung	26.03.24	boe	

Inhaltsverzeichnis

Abbildungsverzeichnis

Referenzen

CiA®-301 v4.2.0 CANopen Application layer and communication profile
CiA®-1301 v5.0.0 CANopen-FD 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
CiA®-1301 v5.0.0 CANopen-FD Application layer and communication profile

1 Übersicht

Der CANopen/CANopen-FD Protokoll Stack stellt grundlegende Kommunikationsmechanismen für eine CANopen/CANopen-FD 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 301/1301 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

- Unterstützung von CANopen Classic und CANopen FD
- Trennung zwischen hardware- abhängigem/unabhängigem Teil mit definierten Interface
- ANSI-C konform
- Einhaltung der MISRA mandatory Regeln
- Unterstützung aller Dienste des CiA-301/1301
- konform zu CiA-301 V4.2 bzw. CiA-1301 V5.0
- 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 Transfer: expedited segmented block	 ● ● -	 ● ● ○	 ● ● ●
USDO Server (CANopen-FD only)	2 sessions	unlimited (255)	unlimited (255)
USDO Client (CANopen-FD only)		unlimited (255)	unlimited (255)
PDO Producer	32	512	512
PDO Consumer	32	512	512
PDO Mapping	statisch	statisch/dynamisch	statisch/dynamisch
MPDO Destination Mode		○	●
MPDO Source Mode		○	●
SYNC Producer		●	●
SYNC Consumer	●	●	●
Time Producer		●	●
Time Consumer		●	●
Emergency Producer	●	●	●
Emergency Consumer		127	127
Guarding Master			●
Guarding Slave	●	●	●
Bootup Handling		●	●
Heartbeat Producer	●	●	●
Heartbeat Consumer	1	127	127
NMT Master-Funktionalität		●	●
NMT Slave	●	●	●
LED CiA-303	●	●	●
LSS CiA-305	●	●	●
Sleep Mode nach CiA-454	●	●	●
Master Bootup CiA-302			●
Configuration Manager			●
Flying Master		○	●
Redundanz		○	○

Dienstmerkmal	Basic Slave	Master/Slave	Manager
Safety (SRDO)	○	○	○
Multiline		○	○
○○○CiA-401 (U8/INT16)			
CiA-4xx	○	○	○

● - inklusive, ○ - optional

3 CANopen Grundlagen

Die nachfolgende Beschreibung von CANopen soll die Grundlagen des Protokolls darlegen, ersetzt aber es nicht das Studium der CANopen Spezifikation CiA 301 4.2 bzw. CiA 1301 V5.0 und weiterer CiA-Spezifikationen.

3.1 Einführung

CANopen ist ein auf CAN basierendes Kommunikationsprotokoll, welches seit 1994 vom CAN in Automation e.V. spezifiziert und gepflegt wird , international in der EN 50325-5 standardisiert ist und seit 2011 in der Spezifikation CiA 301 in der Version 4.2 vorliegt. Dabei werden sowohl Kommunikationsmechanismen als auch Gerätefunktionalitäten definiert.

3.2 CAN als Grundlage

Die Grundlage der CANopen Datenübertragung ist das CAN-Protokoll entsprechend ISO 11898. CANopen nutzt dabei die klassischen CAN-Telegramme mit bis zu 8 Byte Datenlänge und einer Bitrate von bis zu 1 Mbit/s.

CAN-Telegramme bestehen aus

- einer CAN-ID mit 11 bit oder 29 bit,
- Längeninformationen (Data Length Code),
- Nutzdaten von 0 bis 8 Bytes,
- einer CRC und weiteren Sicherungsmechanismen.

Der CAN-Standard (ISO 11898) deckt dabei die beiden unteren Schichten des ISO-OSI-Schichtenmodels, d.h. die Bitübertragungschicht (physical layer) und die Sicherungsschicht (data link layer) ab.

Die CANopen-Spezifikation definiert Regeln zur Verwendung der CAN-ID und Nutzdaten um Applikationsdaten, Kommandos und Nachrichten zur Netzwerküberwachung in definierter Art und Weise zu übertragen.

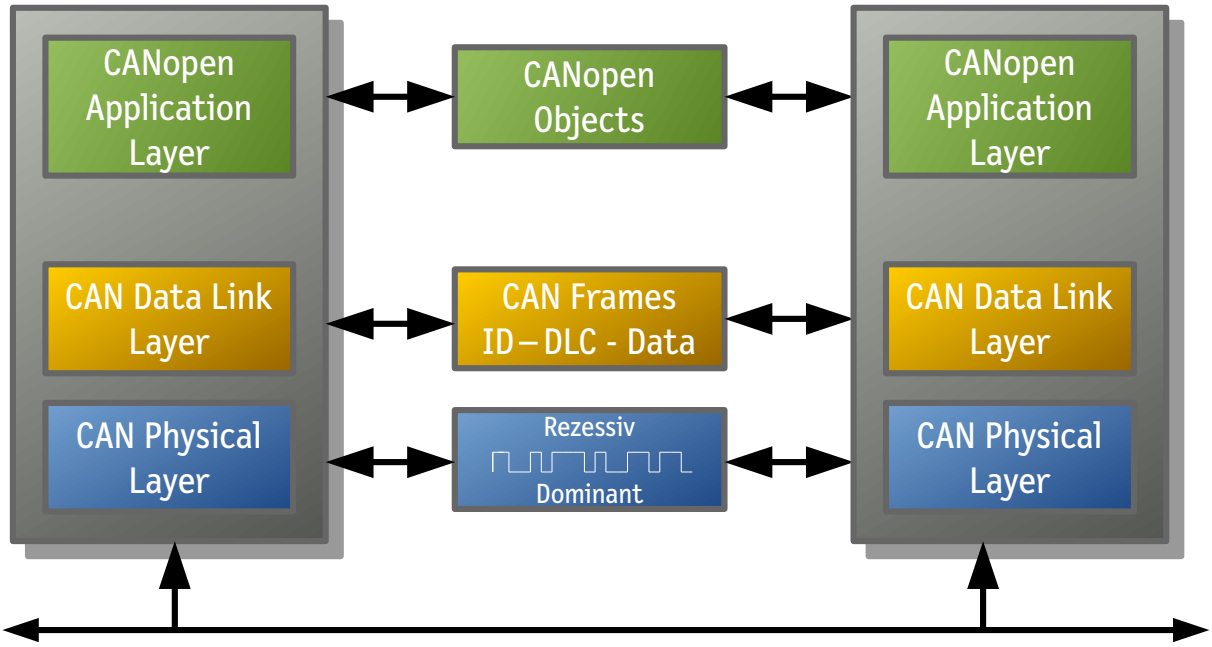
Ein CAN-Netzwerk besteht üblicherweise aus einer Busleitung mit mehreren kurzen Stichleitungen und jeweils 120 Ohm Abschlußwiderständen an beiden Enden.

3.3 CAN-FD als Grundlage für CANopen FD

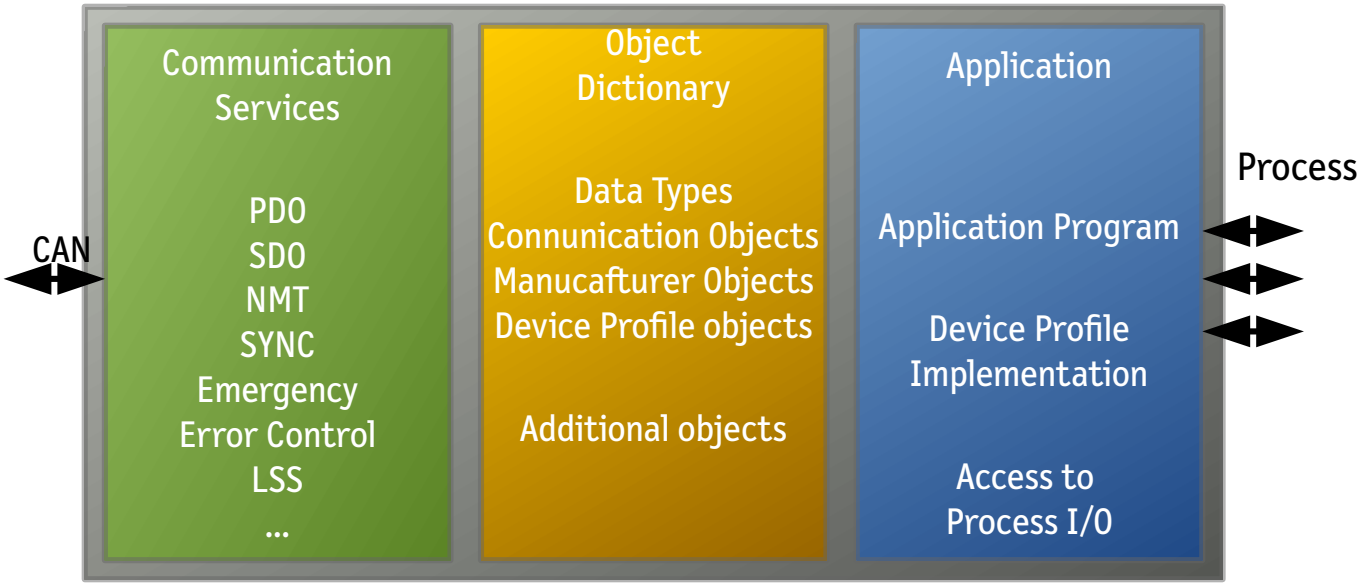
CAN FD als Nachfolger von CAN wird für CANopen FD verwendet. Dabei können die Nutzdaten im CAN FD-Telegramm bis zu 64 Bytes betragen und die Bitrate kann bis zu 8 Mbit/s erreichen. Jedoch ist CAN FD nicht mit klassischem CAN kompatibel. **D.h. ältere CAN/CANopen-Geräte können nicht zusammen mit CAN FD/CANopen FD-Geräten betrieben werden.**

3.4 CANopen-Gerätemodel

Aus CANopen-Sicht tauschen CANopen-Geräte CANopen-Objekte aus, welche wieder um als CAN-Telegramme auf dem CAN-Bus oder als Spannungspegel auf der physikalischen Schicht abgebildet werden. Dabei kann ein CANopen-Objekt auch länger als 8 Byte lang sein. Beispiele dafür sind der CANopen-Gerätename oder z.B. eine komplette Firmware des Geräts - das CANopen-Protokoll sorgt dafür, dass das gesamte Objekt korrekt über CAN übertragen wird.



CANopen-Geräte bestehen aus einer Applikation, einem Objektverzeichnis und den CANopen-Kommunikationsdiensten, welche im CANopen-Stack zusammengefasst sind. Die Applikation hat dabei Zugriff auf I/O-Schnittstellen des Geräts und die Applikation würde in vielen Fällen auch ohne CANopen funktionieren. Wenn nun jedoch die Applikation Daten in einem CANopen-Netzwerk austauschen soll, über CANopen parametriert werden soll und ggf. sogar die Firmware der Applikation über CANopen ausgetauscht werden soll, so ist es erforderlich, dass CANopen Kommunikationsdienste (in Form eines CANopen Stacks) in die Firmware des Geräts integriert werden. Das sogenannte Objektverzeichnis (object dictionary) ist dabei die Schnittstelle zwischen dem CANopen-Stack und der Applikation. Im Objektverzeichnis sind alle Daten der Applikation hinterlegt, welche über CANopen übertragen werden sollen.



In einem CANopen-Netzwerk können bis zu 127 CANopen-Geräte vorhanden sein und CANopen-Geräte haben Knotennummer im Bereich von 1 bis 127. Die Knotennummer kann fest definiert sein, variabel einstellbar (z.B. über Drehschalter oder internen Speicher) oder optional dynamisch zugewiesen werden.

3.5 Objektverzeichnis (OD)

Das Objektverzeichnis ist die Übersicht aller Objekte in einen CANopen-Gerät. Jedes Objekt wird über einen 16-bit Index adressiert und hat einen 8-bit Subindex. Jeder Eintrag im Objektverzeichnis ist sowohl über CANopen – unter Berücksichtigung der Zugriffsrechte - als auch von der Applikation aus zugreifbar. Mit dem emotas CANopen Stack können die Einträge im Objektverzeichnis sowohl reale Variablen in der Applikation als auch vom Stack verwaltete Speicherbereiche sein.

Entsprechend des 16-Bit Index ist das Objektverzeichnis in verschiedene Segmente unterteilt:

Indexbereich	Objekte
0x0000	reserviert
0x0001 - 0x009F	Datentypen
0x00A0 - 0x0FFF	reserviert
0x1000 - 0x1FFF	Communication Profile Segment
0x2000 - 0x5FFF	Manufacturer Specific Profile Segment
0x6000 - 0x9FFF	Standardised Device Profile Area für bis zu 8 Geräteprofile (je 0x800 mögliche Objekte)
0xA000 - 0xAFFF	Prozessabbild für CANopen-PLC (CiA 405)
0xB000 - 0xBFFF	Prozessabbild für CANopen-CANopen-Gateway
0xC000 - 0xFFFF	reserviert

Im Geräteprofilsegment (0x6000-0x9fff) liegen die Daten von bis zu 8 verschiedenen CANopen-Geräteprofilen. Im Herstellerspezifischen Segment (0x2000-0x5fff) können herstellerspezifische Daten abgelegt werden und im Kommunikationssegment (0x1000-0x1fff) liegen Daten zur Konfiguration der CANopen-Kommunikation selbst.

Jedes Objekt kann unterschiedliches Objektcodes und Attribute haben:

Attribut	Beschreibung und Wertebereich
Object Code	Variable, Array, Record, Domain
Data Type	Unsigned8, Unsigned16, Real32, ..
Access	read only (ro), write only (wo), constant (const), read write (rw), *read write write (rww), read write read (rwr)*
Category	mandatory, optional, conditional
PDO Mapping	no, optional, default
Value Range	Wertebereich des Objekts, Prüfung bei SDO-Schreibzugriff
Default Value	Defaultwert nach Initialisierung

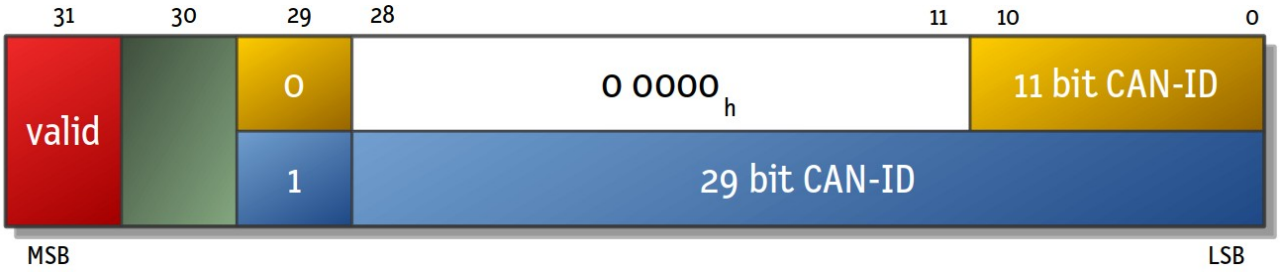
Ein CANopen-Objekt vom Objektcode VARIABLE entspricht in etwa einer C-Variable, d.h. einen singulärem Wert von einem Datentyp. Eine ARRAY ist ein strukturierter Datentyp mit mehreren Elementen des gleichen Datentyps, wiederum ähnlich wie in C und ein RECORD entspricht einem struct in C mit mehreren Elementen unterschiedlicher Datentypen. Eine DOMAIN ist ein unstrukturierter Speicherbereich – wie beispielsweise der Speicher zum Ablegen einer neuen Firmware.

Über CANopen kann man mit den Diensten SDO, (USDO), PDO, MPDO und SRDO auf alle oder eine Teilmenge der Objekte zugreifen. Aus der Applikation erfolgt der Zugriff beim emotas CANopen Stack entweder über typsichere API-Funktionen (z.B. co0dGetObj_i16) oder direkt auf die Applikationsvariable (z.B. gMyValues.TemperatureSensors[3]).

3.6 Kommunikationsobjekte (COB)

Kommunikationsobjekte (COB) werden bei CANopen zum Datenaustausch verwendet und diese Kommunikationsobjekte werden im Kommunikationssegment des Objektverzeichnisses konfiguriert. Aus der Sicht eines Gerätes nutzen bestätige Dienste (z.B. SDO) zwei Kommunikationsobjekte mit 2 verschiedenen CAN-IDs für und unbestätigte Dienste (z.B. PDO, Emergency) nur ein ein Kommunikations-objekt mit einer CAN-ID zur Datenübertragung.

Die COB-ID ist ein Parameter vom Datentyp UNSIGNED32, welcher die jeweilige CAN-ID und 3 weitere Bits zur Konfiguration des Kommunikationsobjekts enthält.



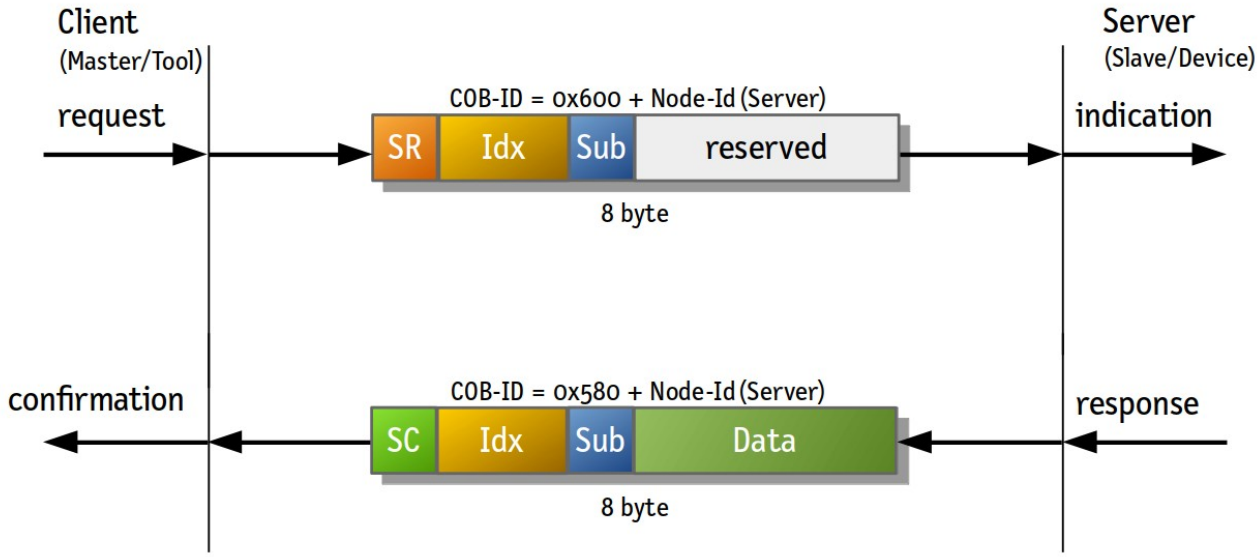
Das höchstwertige Bit (31) definiert bei den meisten Diensten, ob die COB-ID gültig(0) oder deaktiviert(1) ist, und das 29. Bit definiert, ob eine 11-bit CAN-ID oder 29-bit CAN-ID verwendet wird. Die Bedeutung des Bit 30 je nach Kommunikationsdienst unterschiedlich.

Zum Ändern der COB-ID eines Dienstes aus der Applikation heraus kann beim emotas CANopen Stack die API-Funktion coOdSetCobId() verwendet werden.

3.7 Service Data Object (SDO)

Service-Datenobjekte (SDOs) ermöglichen den wahlfreien Zugriff auf alle Objekte des Objektverzeichnisses, werden jedoch vorrangig zur Konfiguration des Gerätes benutzt. Ein CANopen-Slave-Gerät beinhaltet meinst einen oder mehrere CANopen-Server und ein CANopen-Master zusätzlich einen oder mehrere SDO-Clients. In einem SDO-Zugriff kann jeweils nur auf einen Subindex eines Objekts zugegriffen werden.

Ein SDO-Client initiiert einen SDO-Transfer zu einem SDO-Server und der SDO-Server antwortet mit den Daten aus dem lokalen Objektverzeichnis des Servers. Dabei wird das Schreiben zu einem Gerät SDO-Download genannt und das Lesen von einem Gerät ist ein SDO-Upload. Je nach Länge des zu übertragenden Objekts gibt es unterschiedliche SDO-Transfers. Der expedited SDO-Transfer kann nur 4 Byte an Nutzdaten übertragen. Dagegen können mit dem segmentierten Transfer bis zu 4 GB übertragen werden und mit dem SDO-Blocktransfer ebenfalls bis zu 4 GB in schnellerer Art und Weise. Der emotas CANopen Stack unterstützt alle SDO-Transfers und die API ist identisch. Der Stack wählt dabei die jeweils geeignete Transferart entsprechend der Länge des Objekts und dem Funktionsumfang des Kommunikationspartners aus.



Hat ein CANopen-Slave nur einen SDO-Server, so nutzt dieser die Default-COB-Ids für den 1. SDO-Server. Diese sind:

- 0x600 + Node-ID des Servers für die Kommunikation vom Client zum Server
- 0x580 + Node-ID des Servers für die Kommunikation vom Server zum Client

Da bei CAN jede CAN-ID nur von einem einzigen Gerät verwendet werden darf, darf jeweils nur ein SDO-Client (Master/Tool) auf einen SDO-Server zugreifen.

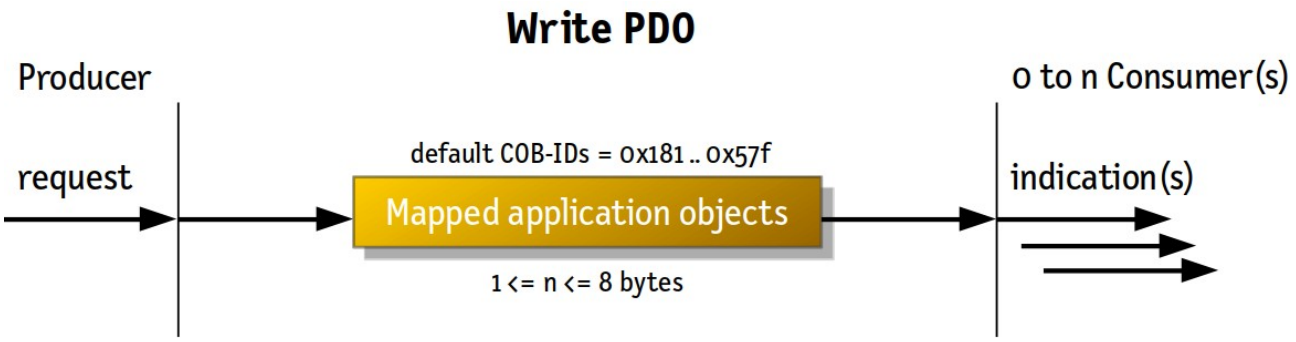
Sind gleichzeitige Zugriffe nötig, so bietet es sich an mehrere SDO-Server in einem Gerät zu implementieren. Für die benötigen CAN-ID-Paare gibt es jedoch keine Vorgaben seitens des CiA e.V., sondern diese sind vom Systemintegrator festzulegen. Die Objekte zur Konfiguration der SDO-Server sind 0x1200 .. 0x127f und der SDO-Clients 0x1280 .. 0x12ff, wobei des Objekt 0x1200, welches den Default-SDO-Server beschreibt optional ist. Die SDO-Kommunikationsparameter in den Objekten haben folgende Struktur:

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

Bei allen SDO-Zugriffen auf Objekte im Objektverzeichnis werden die jeweiligen Zugriffsrechte des Objekts geprüft; bei Schreibzugriffen zusätzlich Datentyp, Größe und ggf. eingeschränkte Wertebereiche. SDO-Transfers können im Fehlerfall von beiden Kommunikationspartner mit einer SDO-Abortnachricht abgebrochen werden. Dafür gibt es in der CANopen-Spezifikation eine große Anzahl von vordefinierter Fehlercodes für alle erdenkliche Fehlerfälle. Der emotas CANopen Stack beantwortet fehlerhafte SDO-Zugriffe weitestgehend selbst mit dem passenden Fehlercode (z.B. Subindex existiert nicht). Darüber hinaus kann die Applikation weitere vordefinierte Fehlercodes wie beispielsweise (0x06060000 Access failed due to an hardware error) zurückliefern.

3.8 Process Data Object (PDO)

Prozessdatenobjekte (PDOs) dienen vorrangig zum Austausch hochpriorer Prozessdaten. Beispielsweise bei einer Batterie mit CANopen-Schnittstelle würde man Strom und Spannung per PDO übertragen und einen Betriebsstundenzähler oder den Herstellernamen per SDO. PDOs werden auf dem CAN als CAN-Nachrichten mit Längen von einem Byte bis zu acht Byte ohne Protokolloverhead in der CAN-Nachricht übertragen. Dabei können mehrere Subindizes unterschiedlicher Objekte aus dem Objektverzeichnis in einem PDO übertragen werden. Die PDO-Kommunikation erfolgt immer im Broadcast von einem PDO-Producer zu einen, keinem oder mehreren PDO-Consumern.



PDOs können synchron oder asynchron gesendet werden. Bei synchronen PDOs wird das Versenden durch eine SYNC-Nachricht getriggert, bei asynchronen PDOs lösen definierte Ereignisse das Senden eines PDOs aus. PDOs, welche von einem Geräte gesendet werden, werden Sende-PDOs (TPDOs) genannt und PDOs, welche von einem Gerät empfangen werden, sind Empfangs-PDOs (RPDOs). Dabei wird immer das jeweilige Gerät betrachten. D.h. dass die TPDOs des einen Geräts, die RPDOs eines anderen Geräts sind.

Ein CANopen-Gerät kann bis zu 512 TPDOs und 512 RPDOs besitzen. In der Praxis haben insbesondere CANopen-Slave-Geräte eher weniger PDOs - oft nur 4 oder wenig mehr. Alle diese PDOs müssen im Kommunikationssegment des Objektverzeichnis konfiguriert werden. Dazu werden pro PDO 2 Objekte benötigt. In den PDO Kommunikationsparameter-Objekten werden die Kommunikationsparameter (CAN-ID, Übertragungsart, Zeiten, ..) definiert und die PDO-Mappingparameter-Objekte beschreiben den Inhalt eines PDOs.

Empfangs-PDOs (RPDO):

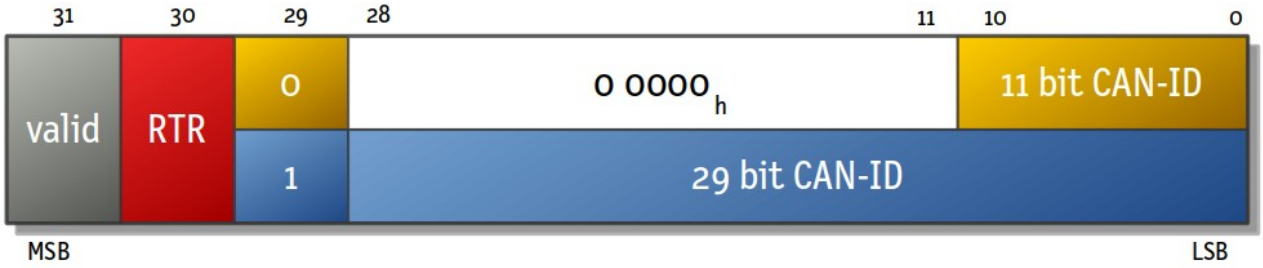
- Kommunikationsparameter: Index 0x1400 .. 0x15FF
- Mappingparameter: Index 0x1600 .. 0x17ff

Sende-PDOs (TPDO):

- Kommunikationsparameter: Index 0x1800 .. 0x19FF
- Mappingparameter: Index 0x1a00 .. 0x1bff

Die Kommunikationsparameter beinhalten die COB-ID (Subindex 1) mit der CAN-ID, die Übertragungsart (Transmission Type) (Subindex 2) und optional eine Sperrzeit (Inhibit time) (Subindex 3), einen Event Timer (Subindex 5) oder einer SYNC-Startwert im Subindex 6.

Die PDO-COB-ID hat folgende Struktur:



Dabei haben die 3 höchstwertigen Bits die folgende Bedeutung:

Feld	Werte	Beschreibung
valid bit	0 1	PDO existiert/ist gültig/wird verarbeitet PDO existiert nicht/ist ungültig/wird nicht verarbeitet
RTR	0 1	PDO kann per RTR abgefragt werden (nur TPDOs) PDO kann nicht per RTR abgefragt werden (nur TPDOs)
frame	0 1	11-bit CAN base frame format 29-bit CAN extended frame format

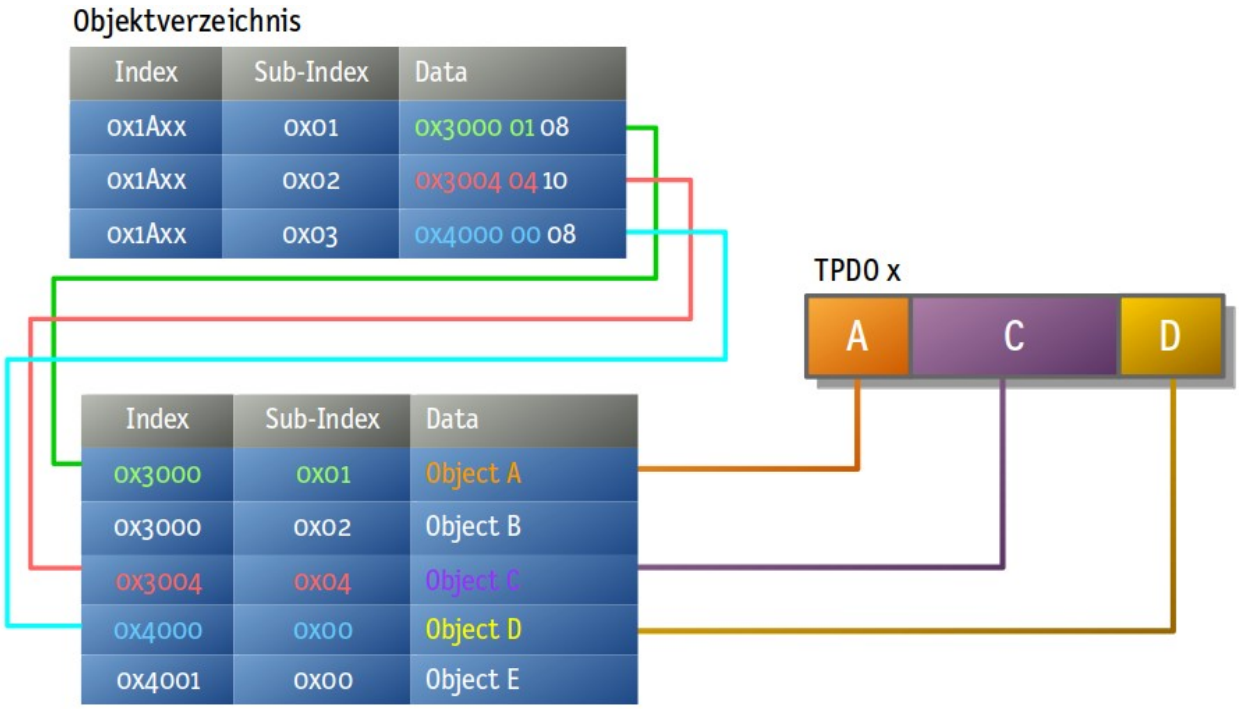
Die möglichen Werte des Transmission Types sind:

Transmission Type	Beschreibung
0	Synchronous acyclic - Übertragung nur bei Änderung nach nächsten SYNC
1 - 240	Synchronous cyclic - Übertragung nach jedem *1. - 240.* SYNC
241 - 251	reserviert
252	Synchronous RTR only - nur bei Anforderung per RTR und SYNC
253	Asynchronous RTR only - nur bei Anforderung per RTR
254	Asynchronous (manufacturer specific) - asynchrone Übertragung
255	Asynchronous (device profile specific) - asynchrone Übertragung nach in Profilen definierten Regeln

Eine Sperrzeit (inhibit time) verhindert das Senden Ereignis-gesteuerter PDOs für ein definiertes Zeitfenster und der Event-Timer sendet Ereignis-gesteuerte PDOs auch dann, wenn kein Ereignis vorlag.

Das PDO-Mapping in den PDO-Mapping-Tabellen (0x1600-0x17ff, 0x1a00-0x1bff) definiert die Zuordnung von Objekten des Objektverzeichnisses in PDOs.

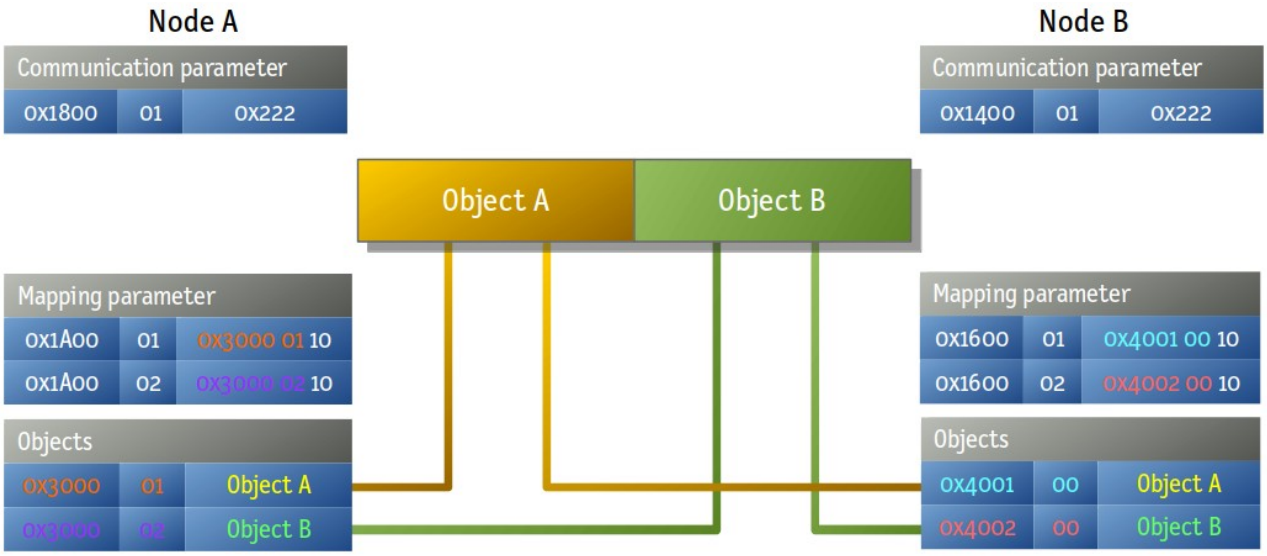
Dabei steht in der Mapping-Tabelle (z.B. Objekt 0x1A00 für TPDO 1) die Objekte, welche in diesem PDO übertragen werden sollen. Im nachfolgenden Beispiel befindet sich im Subindex 1 der Verweis auf Objekt 0x3000, Sub 1, Länge 8 bit, im Subindex 2 der Verweis auf das Objekt 0x30004, Sub 4, Länge 16 bit und im Subindex 3 der Verweis auf das Objekt 0x4000, Sub 0, Länge 8 bit.



Soll nun beispielsweise dieses TPDO 1 z.B. synchron gesendet werden, so geht der CANopen-Stack beim Empfang der SYNC-Nachricht die Mapping-Tabelle durch und kopiert die aktuellen Werte der referenzierten Objekte in das TPDO und sendet das TPDO dann[1]. In dem Beispiel wurden 3 Objekte mit insgesamt 4 Bytes verwendet, aber es können bis zu 8 bytes pro PDO gesendet/empfangen werden. Ist die Mapping-Tabelle eines PDOs zur Kompilierzeit fest und kann nicht mehr geändert werden, so spricht man von statischem Mapping. Ist die Mapping-Tabelle zur Laufzeit über SDO änderbar, so spricht man von dynamischen Mapping. Der emotas CANopen Stack unterstützt beide Varianten.

Als PDO-Linking bezeichnet man das Verknüpfen von PDOs mehrerer Knoten, so dass eine Kommunikationsbeziehung zwischen einem Producer und einem oder mehreren Consumer hergestellt wird. Damit ein PDO von einem Gerät gesendet und von einem anderen Gerät empfangen werden kann, so muss zumindest die CAN-ID der beiden PDOs und die Gesamtlänge des PDOs übereinstimmen. Zudem sollten natürlich die gemappten Objekte in den beiden PDO-Mappingtabellen sinnvoll verknüpft sein.

1 Die tatsächliche Implementierung ist deutlich effizienter. D.h. der Stack hält bereits die Pointer zu den zu sendenden Daten und kopiert diese Werte dann nur in das PDO.

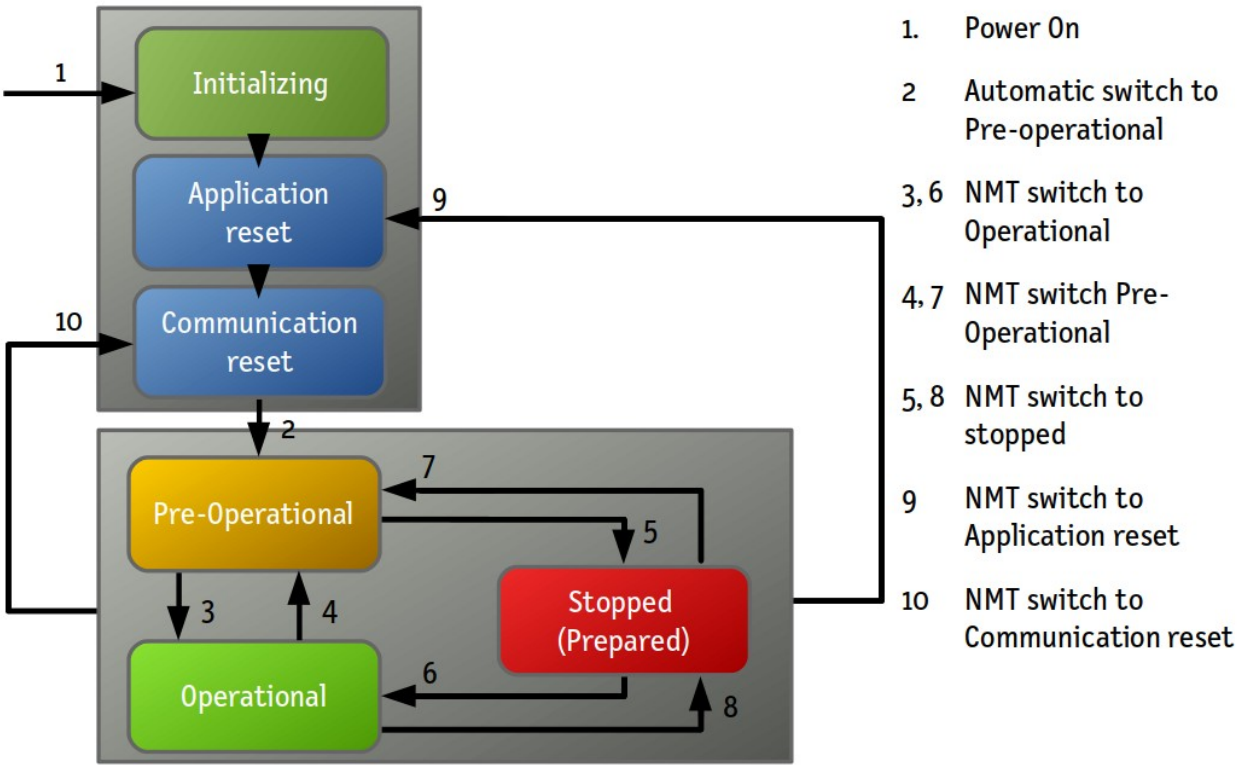


Bei den meisten CANopen-Geräten übernimmt die Konfiguration des PDO-Linkings der Systemintegrator oder das CANopen-Master beim Start des Netzwerks per SDO.

Generell überträgt ein PDO Daten von Objekten eines Geräts in Objekte eines anderes Geräts. D.h. in allen Teilnehmers müssen die Daten in Objekten im Objektverzeichnis vorliegen.

3.9 CANopen Zustandmaschine

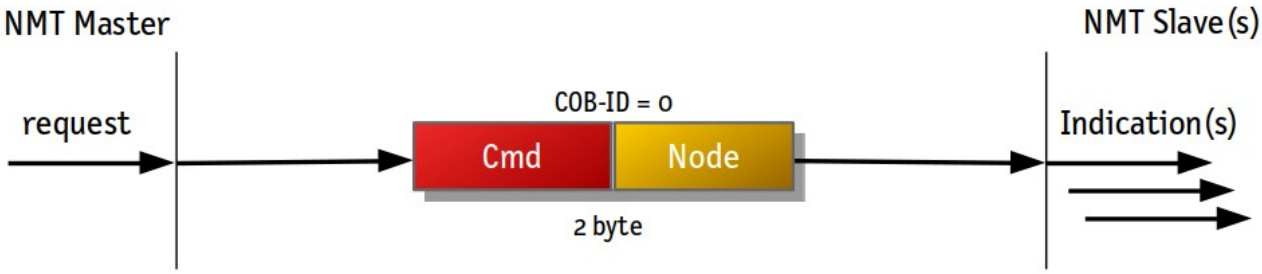
Ein CANopen-Gerät hat einen Zustandsmachine und bestimmte CANopen-Dienste sind nur in bestimmten Zuständen möglich.



Ein CANopen-Geräte geht selbstständig nach der Initialisierung – sofern das Geräte eine Knotennummer hat – in den Zustand „Pre-Operational". Beim Zustandsübergang nach „Pre-Operational" wird vom Gerät eine Boot-Up-Nachricht gesendet. Im Zustand Pre-Operational sind alle CANopen-Dienste außer PDOs möglich und dieser Zustand dient vorrangig zur Konfiguration des Geräts. Durch Kommandos vom NMT-Master wird ein Gerät dann in den Zustand „Operational" gesetzt. Im Zustand „Operation" können alle CANopen-Dienste genutzt werden und dieser Zustand ist der übliche Betriebszustand eines CANopen-Geräts. Im Zustand „Stopped" sind dagegen nur NMT-Kommandos und NMT Error Control-Nachrichten erlaubt. Ist das Objekt 0x1029 nicht implementiert, so wechseln CANopen-Geräte selbstständig im Fehlerfall aus dem Zustand Operational in den Zustand Pre-Operational. Andere automatische Zustandsübergänge in den Zustand Operational sind nur für selbststartende Geräte (self starting devices) erlaubt.

3.10 Network Management (NMT)

Die CANopen Netzwerkmanagement (NMT) – Dienste umfassen Funktionen um die CANopen-Slave-Zustandsmaschine zu steuern.



Der NMT-Master im Netzwerk kann Kommandos zu den einzelnen oder allen CANopen-Slave-Geräten im Netzwerk schicken um die entsprechenden Zustandsübergänge anzuweisen.

Die NMT-Kommandos haben mit der CAN-ID 0x000 die höchste Priorität im CANopen-Netzwerk und bestehen aus 2 Datenbytes. Im ersten Byte ist das Kommando und im zweiten Byte die Zielknotennummer kodiert. Dabei bedeutet eine Knotennummer 0, dass alle Knoten angesprochen werden. Die möglichen Kommandos sind:

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

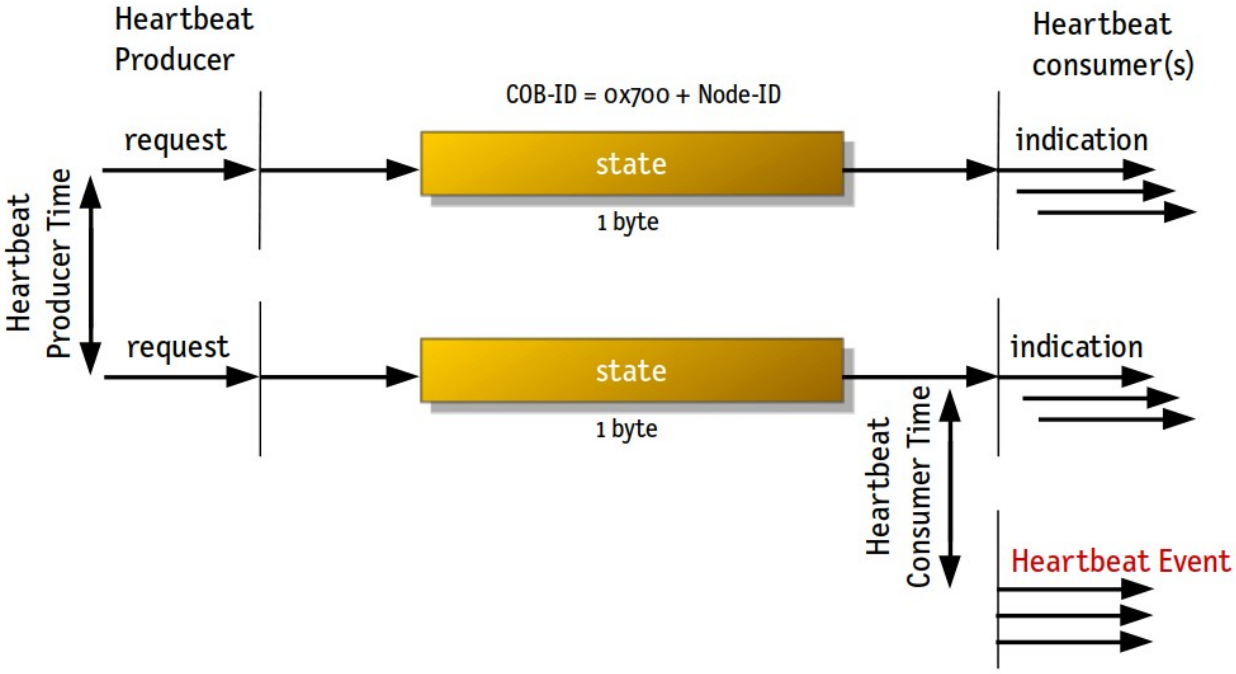
3.11 Knotenüberwachung (ErrCtrl)

NMT Error Control ist der Überbegriff für die Dienste, welche den jeweiligen Zustand eines Knotens anzuzeigen und zu überwachen. Dabei können NMT-Master den Zustand oder Ausfall eines Slaves überwachen, jedoch ist auch eine Überwachung des Master durch einen Slave möglich. Beispielsweise könnte ein Antrieb stoppen, wenn ein Master nicht mehr verfügbar ist.

Die Bootup- und Heartbeat-Nachrichten werden jeweils mit der CAN-ID 0x700+Knotennumer des jeweiligen Gerätes gesendet und übertragen nur jeweils ein Datenbyte. Für einen Boot-up-Nachricht ist der Wert des Datenbytes 0x00 und für Heartbeat-Nachrichten beinhaltet das Datentyp den jeweiligen NMT-Zustand. Gültige Werte sind:

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

Die Heartbeat-Nachrichten werden zyklisch gesendet. Die Zykluszeit des Heartbeats kann für jedes Gerät im Objekt 0x1017 eingestellt werden.

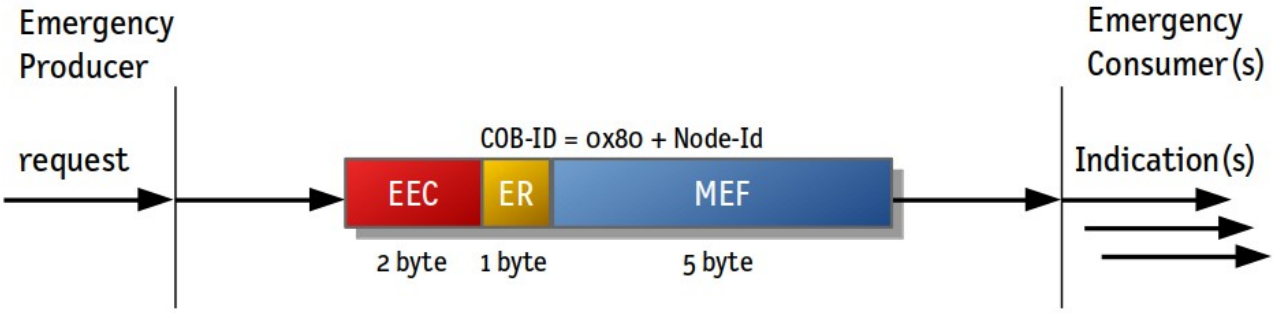


Heartbeat Consumer werden im Objekt 0x1016 konfiguiert. Dabei kann ein Gerät bespielsweise nur einen Heartbeat Consumer Subindes haben, wenn dieses Gerät nur einen Geräte (z.B. den Master) überwachen soll oder alternativ bis 127 Heartbeat Consumer Einträge.

Die älteren Node Guarding und Life Guarding-Dienste werden zwar ebenfalls vollständig durch den emotas CANopen Stack unterstützt, jedoch ist die Verwendung dieser CANopen-Dienste für neue Geräte nicht mehr empfohlen, da diese auf CAN-RTR-Telegrammen aufbauen, deren Verwendung nach CiA AN802 nicht mehr empfohlen ist.

3.12 Emergency (EMCY)

CANopen Emergency (EMCY) Nachrichten sind ein optionaler Bestandteil des CANopen Protokolls um Fehler in den Geräten zu signalisieren. Dennoch ist die Nutzung der Emergency-Nachrichten aus unserer Sicht empfholen. Als Emergeny-Producer werden die Geräte bezeichnet, welche in der Lage sind Emergency Nachrichten zu versenden. Damit können Fehler des Gerätes selbst, der CANopen-Konfiguration und der CAN-Kommunikation signalisiert werden. Die CANopen Spezifikation 301 definiert eine Menge an vordefinierten Fehlercodes und jedes CANopen-Geräte- oder Applikationsprofil kann weitere Fehlercodes definieren. Darüber hinaus ist es auch möglich herstellerspezifische Fehlercodes zu definieren. Zusätzlich können in den Nachrichten bis zu 5 Bytes hersteller-spezifischer Daten übertragen werden. Emergency-Consumer sind die Geräte, welche die EMCY-Nachrichten andere Geräte empfangen können. Dies sind meist nur CANopen-Master-Geräte oder Diagnose- und Konfigurationstools.



Die Länge einer Emergency-Nachricht ist immer 8 Byte. In den ersten beiden Byte wird der jeweilige EMCY Error Code des Fehlers übertragen. Danach folgt der aktuelle Wert des Error Registers (0x1011) und 5 hersteller-spezifische Bytes.

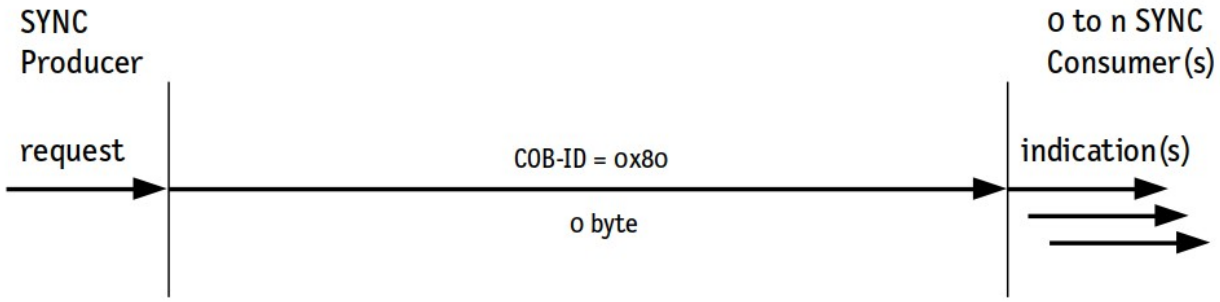
Die CAN-ID ist entsprechend dem Pre-defined Connection Set 0x80 + die Knotennummer des Producers.

Das optionale Inhibit Time Objekt (0x1015) definiert eine minimale Zeitspanne zwischen 2 Fehlernachrichten. Der Defaultvalue des Objekts ist 0 (keine Inhibitzeit) und der Wert kann in Vielfachen von 100µs definitiert werden. Mit einer entsprechenden Konfiguration des Objekts kann der Netzwerk-Master oder Systemintegrator verhindern, dass das CANopen-Netzwerk mit EMCY-Telegrammen überflutet wird.

Bei jedem EMCY-Producer kann optional das Objekt 0x1003 mit 1 bis 254 Subindizes implementiert werden. Ist das Objekt vorhanden, so dient es als Fehlerhistorie, welche die Fehlercodes alle vom Gerät gesendeten. Dabei wird immer der Error Code und ersten beiden hersteller-spezifischen Bytes in die Fehlerhistorie eingetragen, so dass der jeweils neueste Fehler auf Subindex 1 abgelegt wird.

3.13 Synchronisation (SYNC)

Ein Gerät im CANopen-Netzwerk – üblicherweise der Netzwerk-Master – kann SYNC Producer sein und zyklich SYNC-Nachrichten mit der CAN-ID 0x80 und der Länge 0 oder 1 senden.



Die SYNC-Nachrichten werden von den SYNC Consumern empfangen und können zur Steuerung der synchronen PDOs oder zur Synchronisation anderer Aktionen genutzt werden. Mit SYNC-Nachrichten der Länge 1 mit einem Sync Counter kann das Zeitverhalten synchroner PDOs noch detaillierter definiert werden.

3.14 Predefined Connection Set

Das Predefined Connection Set ist eine Vordefinition für COB-Ids für bestimmte Dienste. Laut CiA-Spezifikation müssen unkonfiguräte Geräte dem Predefined Connection Set entsprechen.

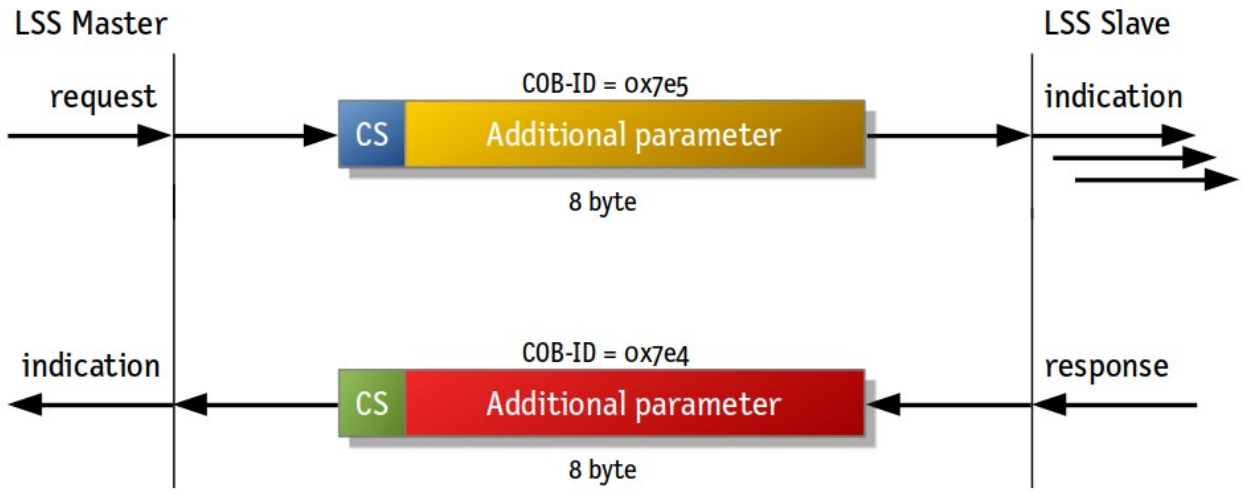
Dienst	COB-ID	Konfigurationsobjekt
NMT	0x000	nicht änderbar
Bootup & Heartbeat	0x700+ Knotennummer	nicht änderbar
SYNC	0x80	0x1005
TIME	0x100	0x1012
Emergency	0x80 + Knotennummer	0x1014
Default-SDO (Client→ Server)	0x600 + Knotennumer (Server)	0x1200 (nicht änderbar)
Default-SDO (Server → Client)	0x580 Knotennummer (Server)	0x1200 (nicht änderbar)
TPDO 1	0x180 + Knotennummer	0x1800
TPDO 2	0x280 + Knotennummer	0x1801
TPDO 3	0x380 + Knotennummer	0x1802
TPDO 4	0x480 + Knotennummer	0x1803
RPDO 1	0x200 + Knotennummer	0x1400
RPDO 2	0x300 + Knotennummer	0x1401
RPDO 3	0x400 + Knotennummer	0x1402
RPDO 4	0x500 + Knotennummer	0x1403

Der emotas CANopen Stack stellt das Predefined Connect Set automatisch nach dem Start des Stacks oder Reset Communication ein. Eine Änderung ist über die Load-Indikation-Funktion des Stacks möglich.

Ein Umkonfiguration der PDO-COB-IDs nach Gerätestart ist üblich, dagegen sollten die COB-IDs für SYNC, Time und Emergency unserer Empfehlung nach eher nicht geändert werden.

3.15 Layer Setting Service (LSS)

Die in CiA-305 definierten Layer Setting Services (LSS) bieten eine Möglichkeit diese Knotennummer dynamisch über CANopen zu vergeben. Zur Adressierung von CANopen-Geräten ohne Knotennummer nutzt der Master die so genannte LSS-Adresse. Diese ist ein 128 bit-Wert, welcher aus den 4 Subindizes des Identity-Objekts (0x1018) besteht: Vendor-Id, Product Code, Revision Number und Serial Number. Dies bedeutet auch dass Geräte, welche eine dynamische Vergabe der Knotennummer unterstützen sollen, eine eindeutige Seriennummer haben müssen.



Grundlegend nutzt das Protokoll nur 2 CAN-Identifier: die CAN-ID 0x7e5 für die Telegramme vom Master zum Slave und die 0x7e4 für die Antworten der Slaves. Alle weiteren unterschiedlichen Kommandos werden anhand des Command Specifiers im ersten Datenbyte der CAN-Nachricht unterschieden.

Um einzelne Geräte umkonfigurieren zu können, müssen diese in den Konfigurationsmodus versetzt werden. Dies ist mit den ‚Switch Mode'-Kommandos möglich. Switch Mode Global wechselt den Zustand aller Geräte im Netzwerk. Dies ist in der Praxis nur zum Verlasen des Konfigurationszustands sinnvoll oder wenn nur 1 Gerät im Netzwerk vorhanden ist.

Soll dagegen ein Gerät selektiv adressiert werden, so wird Switch Mode Selective verwendet. Dabei muss jedoch die vollständige LSS-Adresse (Vendor-ID, Product Code, Revision Number, Serial Number) bekannt sein. Oft ist die LSS-Adresse dem Master oder dem Anwender nicht bekannt. Für diesen Fall – nachdem beispielsweise ein Servicetechniker ein Gerät ausgetauscht hat – stehen 2 Dienste zur Verfügung um unbekannte Geräte zu finden.

Identify Remote Slave (klassischer LSS Scan)

Für diesen Dienst ist es erforderlich, dass Vendor-ID und Produktcode bekannt sind. Jedoch bietet der Dienst die Möglichkeit nach Geräten mit unbekannten Revisions- und Seriennummern zu suchen. Die Anfrage besteht aus 6 CAN-Nachrichten, wobei der Master fragt ob ein Gerät mit einer definierten Vendor-ID und definiertem Produktcode vorhanden ist. Zusätzlich wird eine untere Grenze für die Revisionsnummer und eine obere Grenze für die Revisionsnummer angegeben und das gleiche nochmal für die Seriennummer. Wenn nun auf diese Anfrage ein oder mehrere Geräte antworten, so kann der Master sicher sein, dass es ein oder mehr Geräte gibt, auf diese die Eigenschaften zutreffen. Durch eine geeignete Verkleinerung der Revisionsnummer- und Seriennummerbereiche auf letztlich einen einzelnen Wert, kann mit einer Vielzahl solcher Abfragen ein einzelnen Gerät identifiziert werden.

Dadurch, dass der Master die 6 CAN-Telegramme mit gleicher CAN-ID bei der Anfrage direkt nacheinander sendet, kann es bei Fremdgeräten mit langsamer CAN-Verarbeitung zu Empfangsproblemen und Nachrichtenverlust kommen.

LSS Fast Scan

Mithilfe des neueren LSS Fast Scans können auch Geräte mit unbekannter Vendor-ID und unbekanntem Produktcode identifiziert werden. Zudem besteht beim Fast Scan die Anfrage immer nur aus jeweils einer

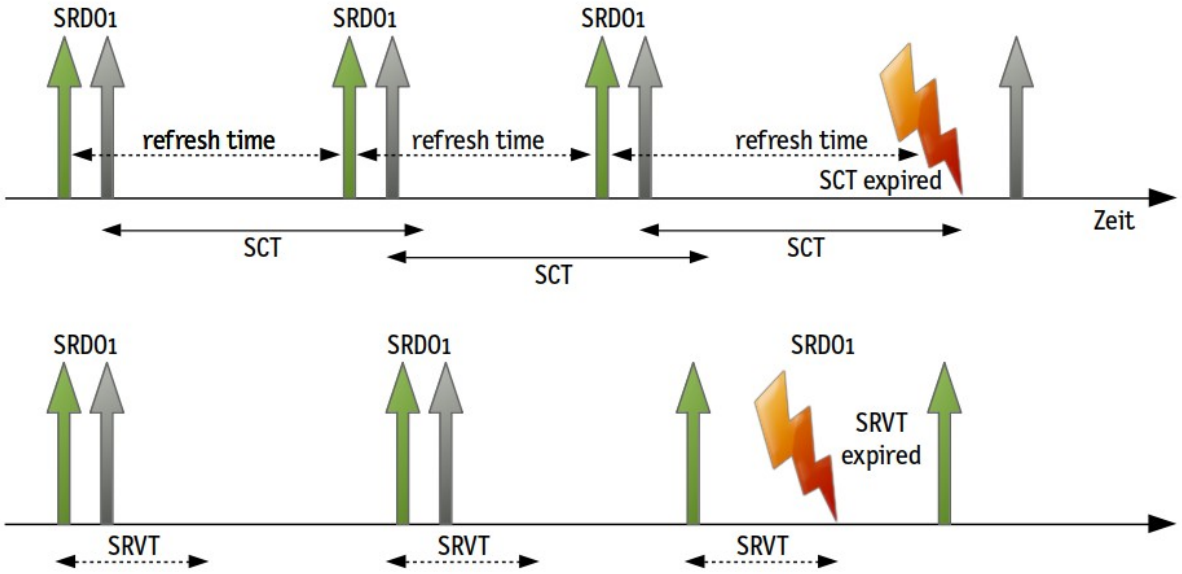
CAN-Nachricht. Dabei fragt der Master die Vendor-ID, Product-Code, Revision-Number und Seriennummer bitweise ab. Dies bedeutet er fragt, ob ein bestimmtes Bit in dem jeweiligen Wert gesetzt ist und daraufhin antworten die Geräte oder nicht und danach fragt der Master das nächste Bit ab und so weiter. Am Ende eines Vorgangs, welcher 4×32 Nachrichten dauert, ist mindestens 1 Gerät identifiziert. Danach kann der Master den Vorgang wiederholen um weitere Geräte zu finden. Mittels Fast Scan kann der Master auch bekannt Geräte direkt adressieren.

Da die Vorteile von LSS Fast Scan überwiegen ist dies für neuere Geräte empfohlen bzw. von diversen CiA-Applikationsprofilen gefordert. Der emotas CANopen Stack beinhaltet beide Verfahren.

3.16 Safety Relevant Data Object (SRDO)

Neben den herkömmlichen CANopen-Diensten zur Datenübertragung wie SDO und PDO wird für die Übertragung sicherheitsrelevanter Daten nach SIL3 mit dem SRDO-Dienst (Safety Related Data Object) ein spezieller Dienst für diese Datenübertragung definiert. Hinsichtlich der Konfiguration und Art der Kommunikation ähnelt diese einem PDO, jedoch werden zusätzliche Eigenschaften definiert. Diese sind:

- zyklische Datenübertragung mit Timeout-Überwachung der Safety Cycle Time (SCT)
- Zweifache Übertragung der Nutzdaten, davon einmal bitweise invertiert
- Prüfung den Datenkonsistenz
- Prüfung des zeitlichen Abstands der invertieren und nicht-invertierten Daten innerhalb der Safety Relevant Validation Time (SRVT)
- Sicherung der Konfiguration durch eine CRC.



Der SRDO-Support ist eine optionale Erweiterung zum emotas CANopen Stack und wird in einem separaten Safety-Handbuch beschrieben.

3.17 Unterschiede und Neuerungen bei CANopen FD

Die Grundprinzipien von CANopen FD und CANopen sind gleich. Änderungen bestehen hinsichtlich folgender Punkte:

- Länge der PDOs auf 64 Byte erweitert
- keine RTR-Nachrichten mehr in CAN FD: keine PDO-Abfrage per RTR, kein Node Guarding

- kein bitweises PDO-Mapping
- Emergency-Nachrichten verlängert mit deutlich mehr Informationen
- SDO wurde durch mächtigeren USDO-Dienst ersetzt
- *Mai 2020: Zusatzdienste wie LSS oder SRDO noch nicht für CANopen FD definiert*

Mit dem emotas CANopen FD Stack kann zur Laufzeit beim Start des Stacks CANopen oder CANopen FD ausgewählt werden.

Der USDO Service in CANopen-FD ist die Weiterentwicklung des SDO Service aus CANopen. Er ist primär für Konfigurations- und Diagnosedaten konzipiert. Durch die Verwendung von CAN-FD können Telegramme bis zu 64 Byte lang sein. Für die Übertragung stehen das Expedited, Segmented und Bulk Protokoll zur Verfügung. Daten können per USDO sowohl in Unicast als auch Broadcast gesendet werden. Für Netzwerkübergreifende Transfers steht ein eigener Remote Service zur Verfügung.

4 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

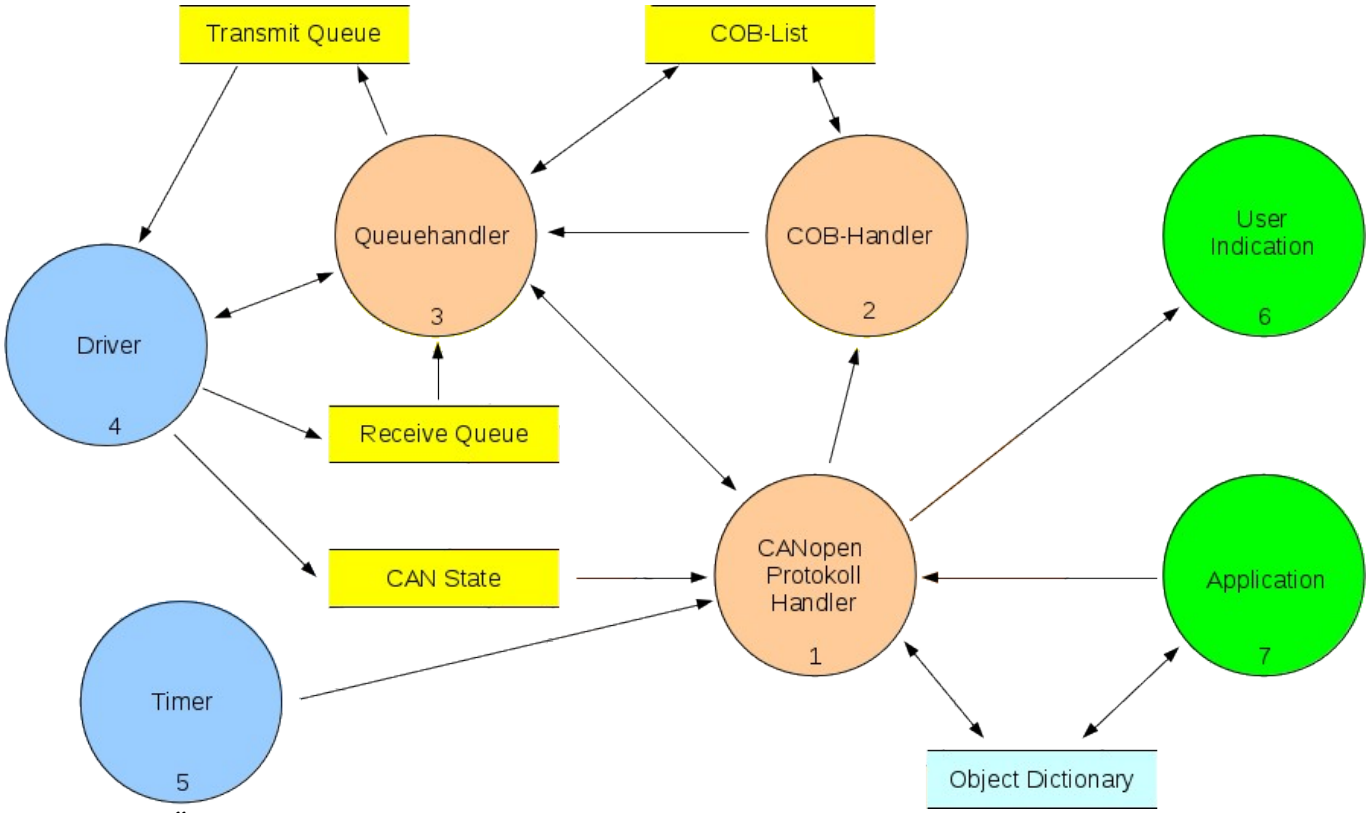


Abbildung 1: Überblick über die Module

Die 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 Aufführungsstatus 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 6).

Das User-Interface zwischen CANopen und CANopen FD ist identisch und unterscheidet sich nur in wenigen Funktionen:

5 CANopen classic und CANopen FD

Das User-Interface zwischen CANopen und CANopen FD ist identisch und unterscheidet sich nur in wenigen Funktionen:

	CANopen classic (Single/Multiline)	CANopen FD (Single/Multiline)	CANopen classic + FD (Multiline)
SDO Funktionen/Indikation	Vorhanden	-	Vorhanden
USDO Funktionen/Indikation	-	Vorhanden	Vorhanden
EMCY Producer	3 Parameter	6 Parameter	6 Parameter
EMCY Consumer	4 Parameter	9 Parameter	9 Parameter

Solange die Funktionalität aller CAN-Linien gleich ist (CANopen classic oder CANopen FD), gelten die jeweiligen Funktionsparameter. Wird bei einer Linie CANopen FD genutzt, gelten die CANopen FD Funktionsparameter für alle Linien, auch wenn die Linie im CANopen Classic Mode betrieben wird. In diesem Fall werden die zusätzlichen Parameter für CANopen FD ignoriert bzw. als 0 übergeben.

Ob CANopen FD Funktionalität prinzipiell unterstützt werden soll, wird im CANopen DeviceDesigner festgelegt. Der CANopen FD Stacks bietet zusätzlich die Möglichkeit, den CAN Mode während der Initialisierung zu ändern. Dazu ist der Funktion
coCanOpenStackInit() als Parameter eine Liste zu übergeben, welche Linien als classic bzw. FD arbeiten sollen.

6 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_UPDATE	Update PDO Daten vor dem Versenden	Index Subindex	
PDO_REC_EVENT	Time Out für PDO	PDO Nummer	

EVENT_TYPE	Event	Parameter	Rückgabewert
MPDO	Multiplexed PDO empfangen	PDO Nummer Index 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 Receive data	Ok/SDO Abort Code
SDO_SERVER_DOMAIN_WRITE	SDO Domain Größe erreicht	Index subindex Domain Buffer Size Transfered 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	
USDO_SERVER_READ	USDO Server Read Transfer beginnt	nodeId index subindex	Ok/SDO Abort Code/ Split Indikation
USDO_SERVER_WRITE	SDO Server Write Transfer beendet	nodeId index subindex	Ok/SDO Abort Code/ Split Indikation
USDO_SERVER_CHECK_WRITE	SDO Server Write Transfer beginnt	nodeId index subindex Pointer to Received data	Ok/SDO Abort Code
USDO_SERVER_DOMAIN_WRITE	SDO Domain Größe erreicht	Index subindex Domain Buffer Size Transfered Size	
USDO_CLIENT_READ	SDO Client Read Transfer beendet	nodeId index subindex Result	
USDO_CLIENT_WRITE	SDO Client Write Transfer	nodeId	

EVENT_TYPE	Event	Parameter	Rückgabewert
	beendet	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	
MANAGER_BOOTUP	Manager Event aufgetreten	NodeID Event Type	
FLYMA	Flying Master Status	State Master Node Priorität	
SRD	SRD Antwort vom Master	Result Fehlercode	
GW_SDOCLIENT_USER	Client SDO für Gateway Funktionalität	-	SDO Nr.

Für jedes 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	Testmode – die Funktion prüft ob die Funktionalität mit den übergebenen Parametern ausgeführt werden könnte. Der Rückgabewert der Funktion wird ausgewertet. Indikation Funktionalität darf noch **nicht** ausgeführt werden
	CO_TRUE	Ausführungsmode – die Funktionalität wird mit den übergebenen Parametern ausgeführt. Der 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.

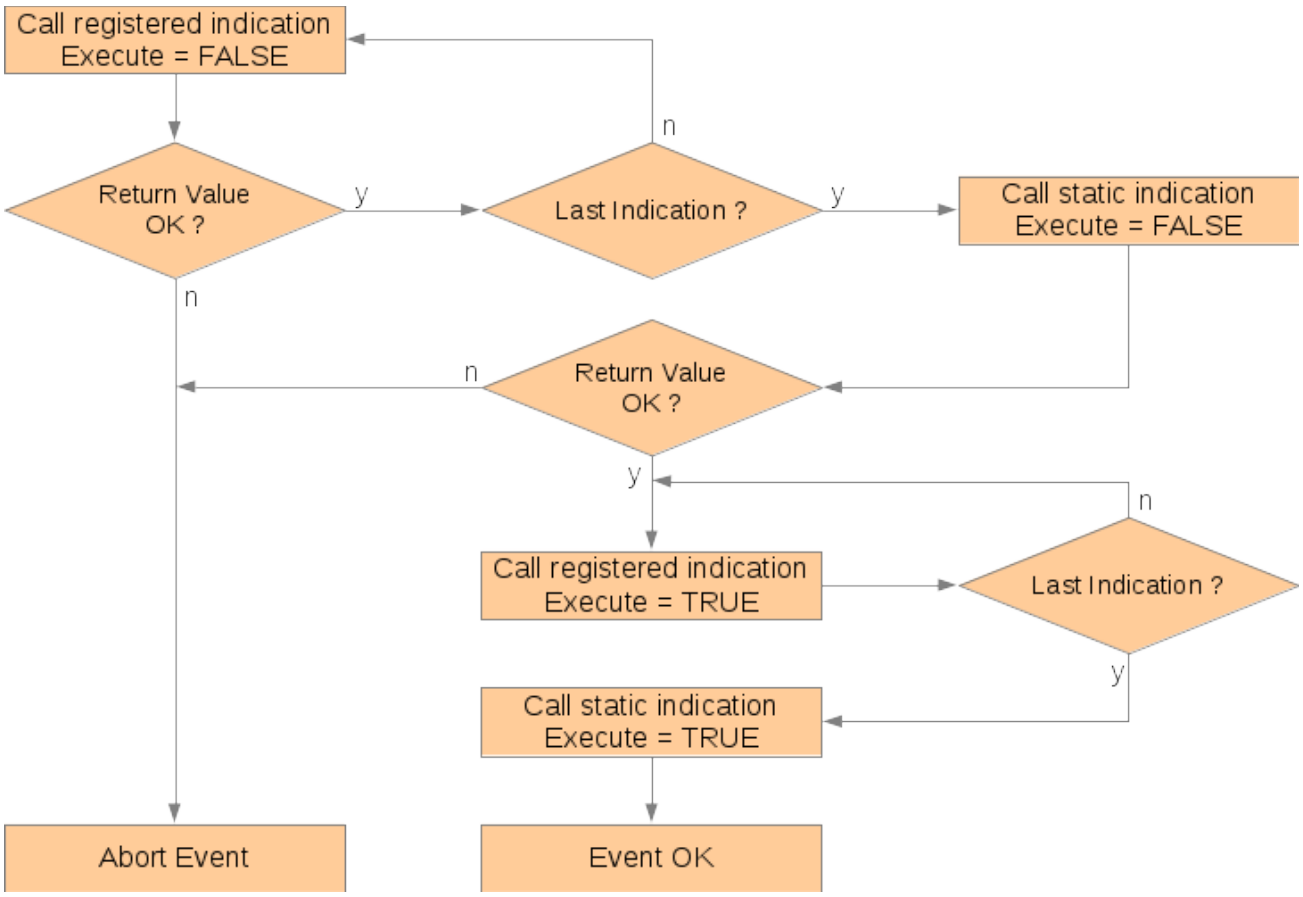


Abbildung 2: Indicationen

7 Das Objektverzeichnis

Das Objektverzeichnis wird mit dem CANopen DeviceDesigner generiert und während der Initialisierung dem CANopen Protokoll Stack übergeben. Dabei gelten die vom CiA 301/1301 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

7.1 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_u32 };
```

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.

7.2 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 beschreibbar
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
CO_ATTR_DYNOD	Objekt ist dynamisch angelegt
CO_ATTR_STORE	Objekt soll nichtflüchtig gespeichert werden
CO_ATTR_COMPACT	Objekt hat identische Subindexe
CO_ATTR_FD	Objekt gilt nur im FD-Mode
CO_ATTR_STD	Objekt gilt nur im classical CAN Mode

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

7.3 Objektverzeichnis Zuordnung

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

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

7.4 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 müssen von der Applikation bereitgestellt werden. Die Zeiger auf die Strings sowie die aktuelle und maximale Stringlänge werden in internen Listen verwaltet. Diese können mit den Zugriffsfunktionen *coOdVisStringSet()* bzw. *coOdSetObjSize()* zur Laufzeit modifiziert werden. Beim Eintragen des Default Strings im *CANopen DeviceDesigner* wird die aktuelle und maximale Länge auf die Länge des eingetragenen Strings gesetzt.

Domains werden durch die Applikation verwaltet. Startadresse, maximale und aktuelle Größe können zur Laufzeit mit den Funktionen *coOdDomainAddrSet()* bzw. *oOdSetObjSize()* gesetzt werden.

Beim Empfang von Strings oder Domains wird das Objekt auf die aktuelle Länge entsprechend der empfangenen Länge gesetzt. Beim Auslesen des Objekts wird dann die aktuelle Länge geliefert. Ein nochmaliges Schreiben ist aber immer bis zur maximalen Datenlänge möglich.

7.4.1 Domain Indikation

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.

7.5 Dynamisches Objektverzeichnis

7.5.1 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 example_sl/dynod.

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

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_sl/dynod_appl genutzt werden.

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

8 CANopen Protokoll Stack Dienste

8.1 Initialisierungsfunktionen

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

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

8.1.1 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 coEventRegister_NMT()) aufgerufen.

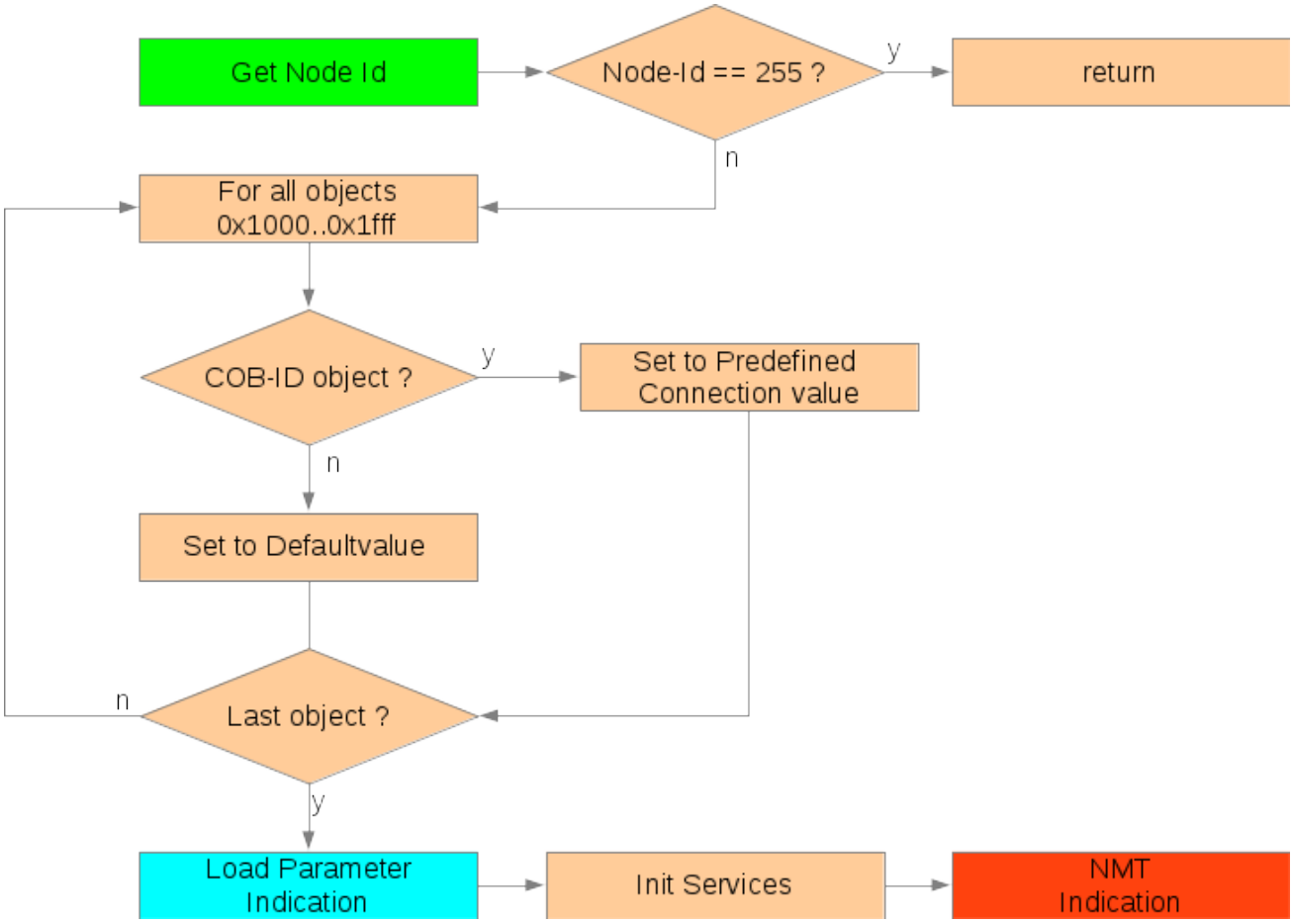


Abbildung 3: Reset Communication

8.1.2 Reset Applikation

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

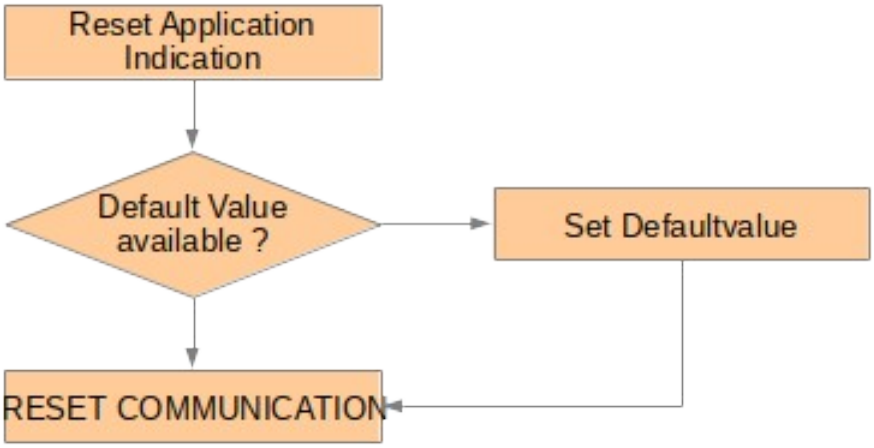


Abbildung 4: Reset Application

8.1.3 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.

8.2 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.

8.2.1 Load Parameter

Beim Aufruf der Reset-Funktionen (Reset Communication, Reset Application) können die Defaultwerte für die Objekte über eine Indikation Funktion überschrieben werden. Die Anmeldung der Indikation Funktion kann automatisch durch die Initialisierungsfunktion coCanOpenStackInit() erfolgen.

Die Indikation Funktion wird dann bei jedem Reset Communikation und Reset Applikation aufgerufen, und soll die mit SAVE Parameter (siehe Abschnitt 8.2.2) 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.

8.2.2 Save Parameter

Die Sicherung von Objekten in einem nicht-flüchtigen Speicher erfolgt nach dem Schreiben auf das Objekt 0x1010 mit dem speziellen Wert „save“ (0x65766173). Dafür ist eine entsprechende Funktion über coEventRegister_SAVE_PARA() zu hinterlegen, die das Speichern im nicht-flüchtigem Speicher 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.

8.2.3 Clear Parameter

Das Löschen der Objektverzeichnis Daten im nicht-flüchtigen Speicher erfolgt nach dem Schreiben auf das Objekt 0x1011 mit dem speziellen Wert „load“ (0x64616f6c). Dafür ist eine entsprechende Funktion über coEventRegister_CLEAR_PARA() zu hinterlegen, die das Löschen in dem nicht-flüchtigem Speicher realisiert.

Bei einem folgenden Reset Applikation oder Reset Communication sollten dann beim Aufruf der Load Parameter Funktion (siehe 8.2.1) keine Daten mehr geladen werden.

8.3 SDO

SDOs sind nur im CANopen classic Mode nutzbar. Im CANopen FD Mode sind dafür die USDO vorgesehen.

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.

8.3.1 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 coEventRegister_SDO_SERVER_READ(), coEventRegister_SDO_SERVER_WRITE() und coEventRegister_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.

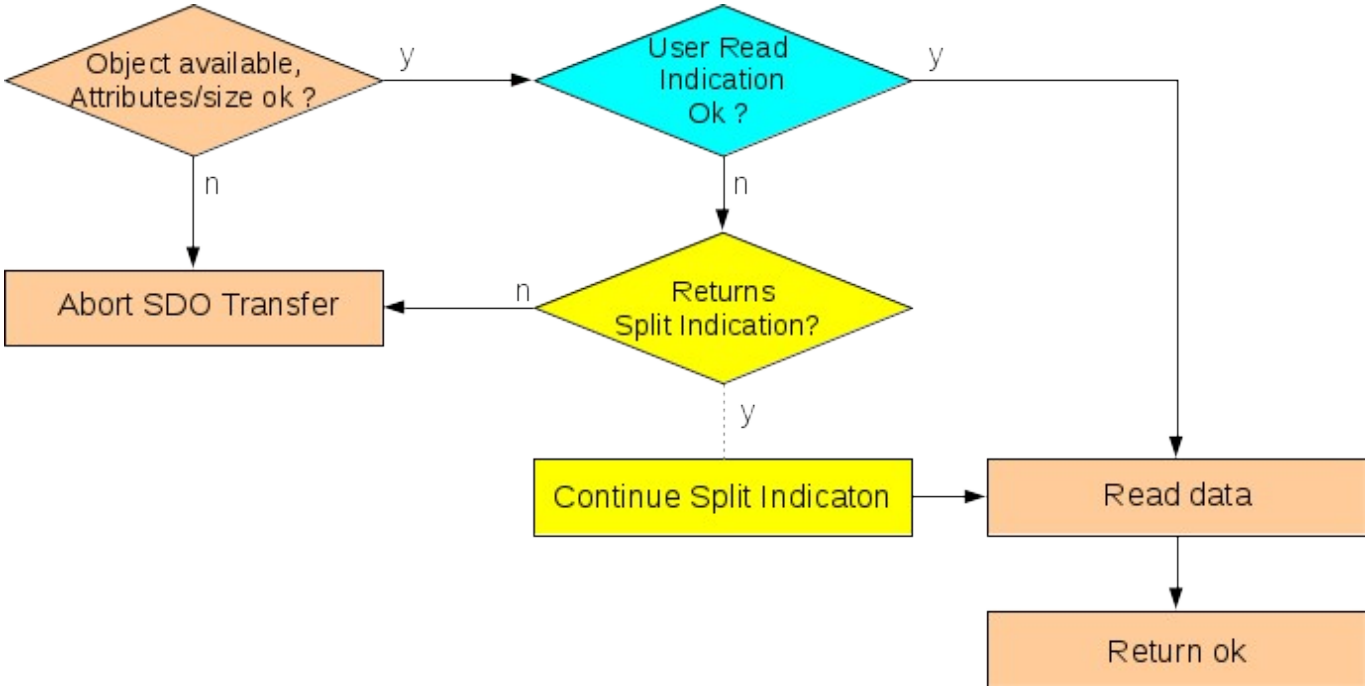


Abbildung 5: SDo Server Read

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.

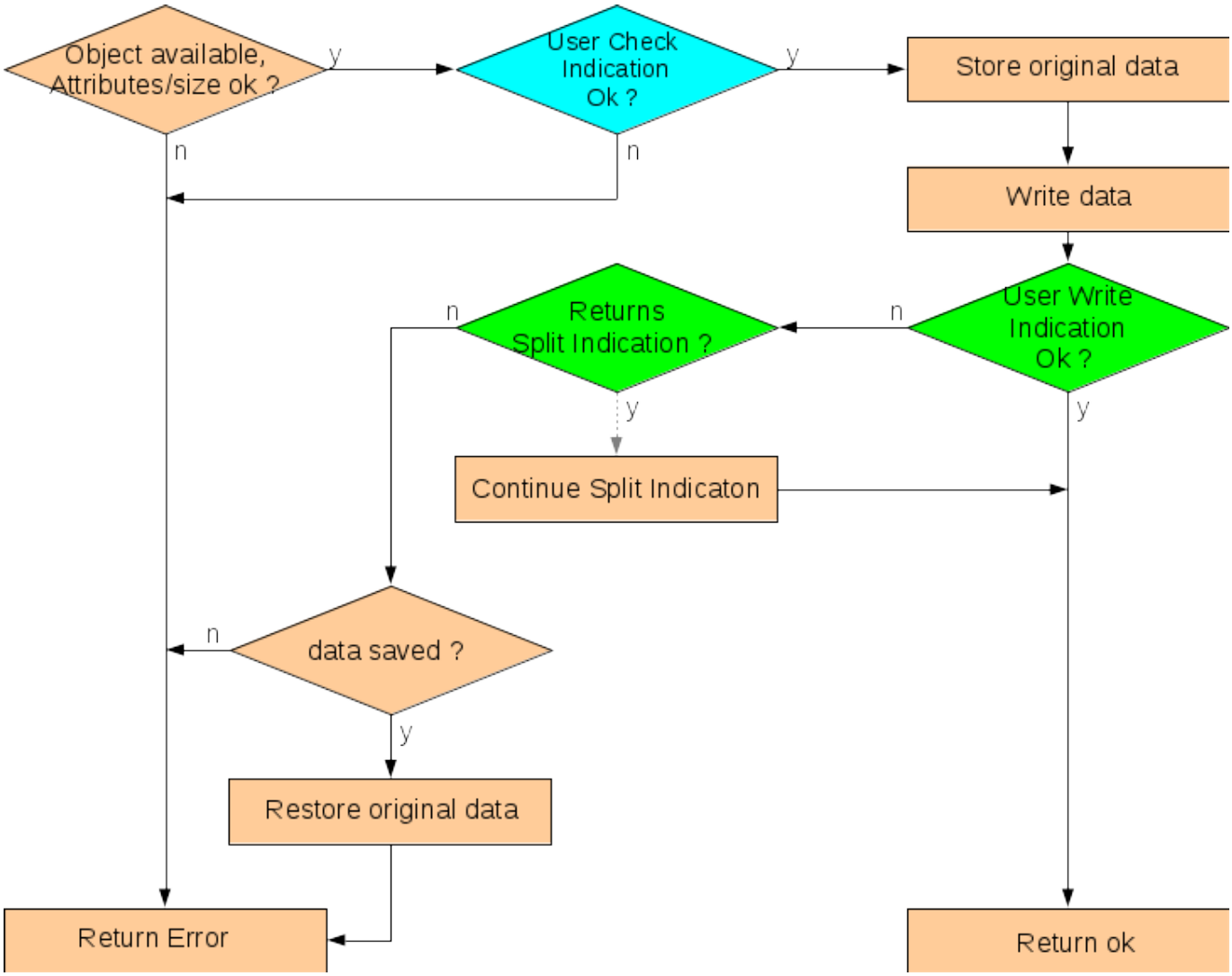


Abbildung 6: SDO Server Write

8.3.2 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 coEventRegister_SDO_CLIENT_READ() und coEventRegister_SDO_CLIENT_WRITE()) mitgeteilt.
Bei jeder Nachrichtenübertragung wird eine Timeout-Überwachung gestartet, die nach Ablauf der eingestellten Zeit die registrierten Event Funktionen (siehe coEventRegister_SDO_CLIENT_READ() und coEventRegister_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.

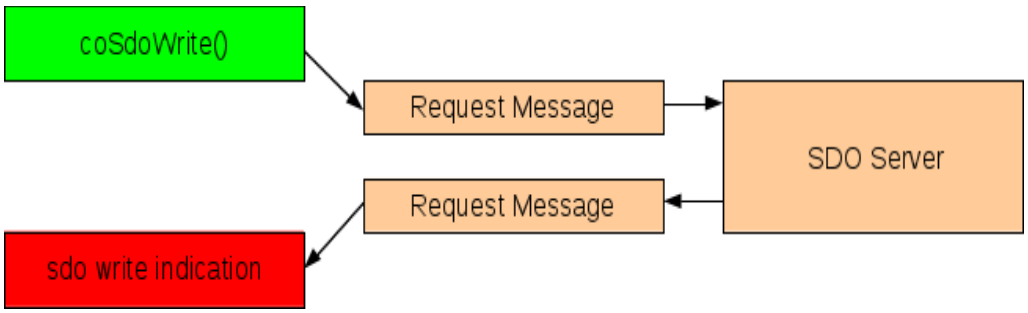


Abbildung 7: SDO Server Write

8.3.3 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.

8.4 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.

8.5 USDO

USDOs sind nur im CANopen FD Mode nutzbar. Die gleichzeitige Nutzung mit SDOs ist nicht möglich.

Die COB-IDs für USDOs werden anhand der eigenen Knotennummer gesetzt und sind nicht änderbar. Aktuell sind keine Objekte für die Konfiguration im Objektverzeichnis notwendig.

Die Konfiguration der USDOs erfolgen im CANopen DeviceDesigner.

8.5.1 USDO Server

USDO 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 coEventRegister_USDO_SERVER_READ(), coEventRegister_USDO_SERVER_WRITE() und coEventRegister_USDO_SERVER_CHECK_WRITE())informiert.

Die Anzahl der gleichzeitig nutzbaren USDO Verbindungen kann im CANopen DeviceDesigner festgelegt werden.

Der USDO 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.

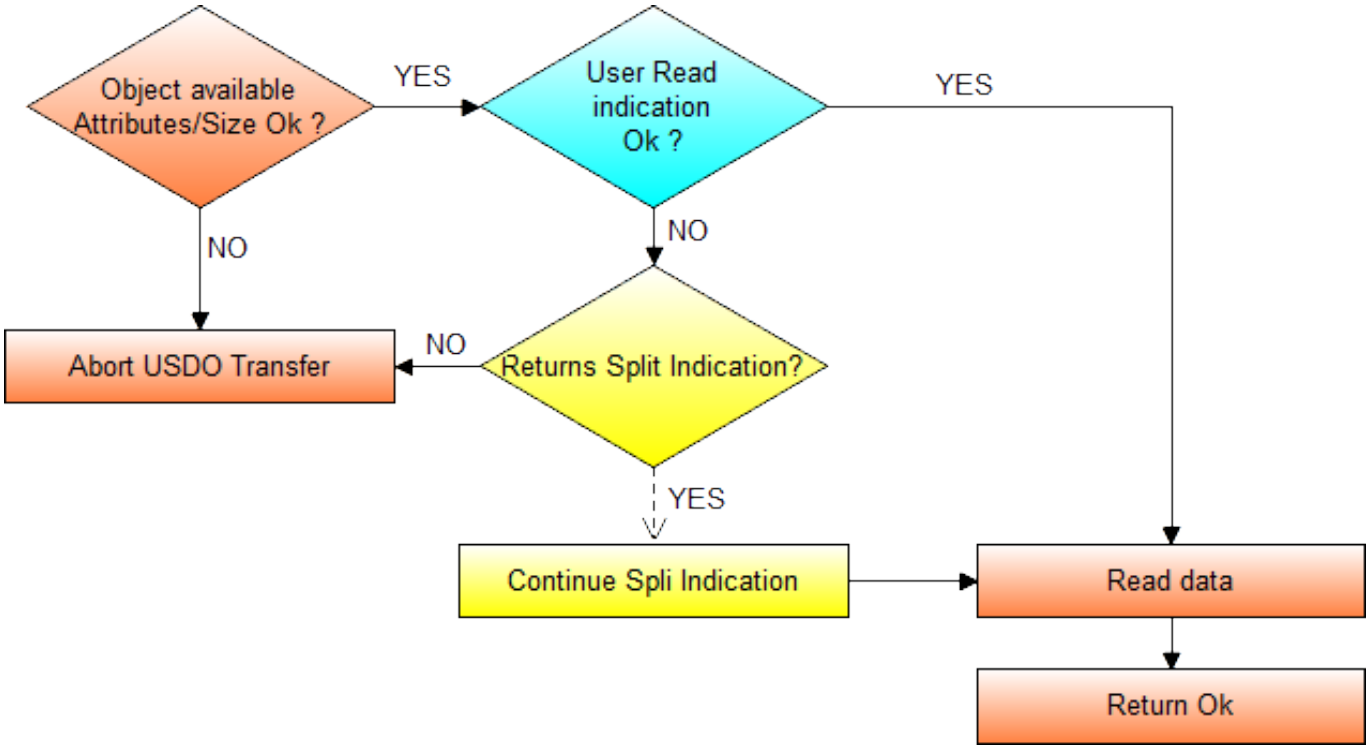


Abbildung 8: USDO Server Read

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.

8.5.2 USDO Client

USDO Client Dienste müssen von der Applikation angefordert werden. Dafür stehen die Funktionen *coUsdoRead()* und *coUsdoWrite()* zur Verfügung. Sie starten den notwendigen Transfer. Asynchron dazu wird der Abschluss der Übertragung über die registrierten Event Funktionen (siehe coEventRegister_USDO_CLIENT_READ() und coEventRegister_USDO_CLIENT_WRITE()) mitgeteilt.
Bei jeder Nachrichtenübertragung wird eine Timeout-Überwachung gestartet, die nach Ablauf der eingestellten Zeit die registrierten Event Funktionen (siehe coEventRegister_USDO_CLIENT_READ() und coEventRegister_USDO_CLIENT_WRITE() bzw. die beim Start der Funktion übergebenen Funktion 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.

Abbildung 9: USDO Client Write

8.6 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 timer-getriebenen 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 coEventRegister_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 coEventRegister_PDO() und coEventRegister_PDO_SYNC()).

8.6.1 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

8.6.2 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 Mappingtabellen bei der Initialisierung und bei der Aktivierung des Mapping (Schreiben auf sub 0) 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 0 der Mappingeinträge auf 0 setzen)
- Mapping Einträge modifizieren
- Mapping einschalten (Subindex 0 der Mappingeinträge auf gewünschte Anzahl setzen)
- PDO einschalten (NO_VALID_BIT in PDO COB-ID rücksetzen)

8.6.3 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 coEventRegister_PDO_REC_EVENT()) aufgerufen.

8.6.4 PDO Daten Update

PDOs nutzen zum Versenden die Daten aus dem Objektverzeichnis. Falls diese Daten vor dem Versenden eines PDOs noch einmal aktualisiert werden sollen, kann dies über die registrierte Indikation Funktion (siehe coEventRegister_PDO_UPDATE()) erfolgen.

8.6.5 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.

8.6.6 PDO und SYNC

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

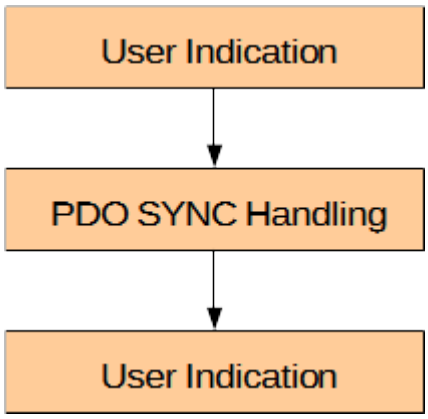


Abbildung 10: PDO Sync

8.6.7 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.

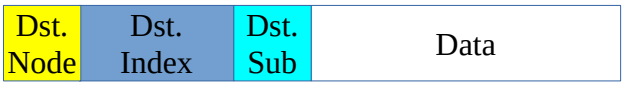
Mit der Funktion register_MPDO() kann eine Indikation Funktion angemeldet werden, die beim Eintreffen eines MPDOs aufgerufen wird.

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.

8.6.7.1 MPDO Destination Address Mode (DAM)

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



8.6.7.1.1 MPDO DAM Producer

Einträge im Objektverzeichnis

Index	Subindex	Beschreibung	Wert
18xx$_h$		PDO Kommunikationsparameter	
1Axx$_h$	0	Anzahl Mapping Einträge	255
1Axx$_h$	1	Mapping Eintrag	Appl spezifisch

8.6.7.1.2 MPDO DAM Consumer

Einträge im Objektverzeichnis

Index	Subindex	Beschreibung	Wert
14xx$_h$		PDO Kommunikationsparameter	
16xx$_h$	0	Anzahl Mapping Einträge	255

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

8.6.7.2 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:

8.6.7.2.1 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
$18xx_h$		PDO Kommunikationsparameter	
$18xx_h$	2	Transmission Type	254/255
$1Axx_h$	0	Anzahl Mapping Einträge	254
$1FA0_h..1FCF_h$	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

8.6.7.2.2 MPDO SAM Consumer

Einträge im Objektverzeichnis

Index	Subindex	Beschreibung	Wert
$14xx_h$		PDO Kommunikationsparameter	
$16xx_h$	0	Anzahl Mapping Einträge	254
$1FD0_h..1FFF_h$	0-254	Dispatcher Liste	

Die Dispatcher Liste 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.

Dispatcher Liste

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.

8.7 Emergency

8.7.1 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 coEventRegister_EMCY) durch den Anwender gesetzt oder auch das Senden verhindert werden.

8.7.2 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 (coEventRegister_EMCY_CONSUMER()) der Applikation als empfangene Emergency-Nachricht zur Verfügung gestellt.

8.8 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 coEventRegister_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.

8.8.1 NMT Slave

NMT Slave Geräte müssen die vom NMT Master gesendeten Nachrichten umsetzen. Empfangene NMT Kommandos können über die Event-Funktion coEventRegister_NMT() der Applikation mitgeteilt werden.

8.8.2 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 Master Knoten gelten soll.

8.8.3 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 (coEventRegister_EMCY()) registriert ist, kann hier die 5 additional Bytes vor dem Versenden modifiziert werden.

8.9 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 Funkionen vorgesehen (siehe coEventRegister_SYNC() und coEventRegister_SYNC_FINISHED()):

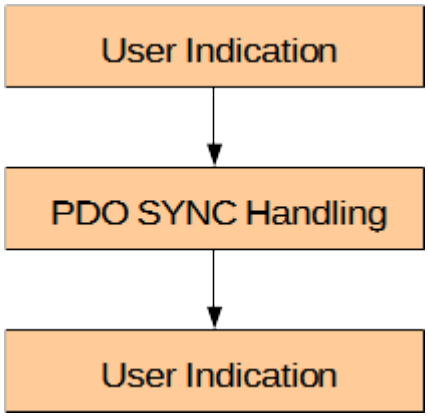


Abbildung 11: SYNC Handling

8.10 Heartbeat

8.10.1 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.

8.10.2 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 coEventRegister_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

8.11 Life Guarding

Das Life Guarding ist nur für CANopen Classical verfügbar.

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 8.8.3 Default Error Behaviour) ausgeführt, und die registrierte Indikation Funktion (siehe coEventRegister_ERRCTRL()) aufgerufen.

8.12 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 coEventRegister_TIME()) angezeigt.

8.13 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 coEventRegister_LED_GREEN() und coEventRegister_LED_RED()) ein- bzw. ausgeschaltet.

8.14 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:

Persistant Node-ID	Power-On Value, wird durch die Applikation bereitgestellt
Pending Node-ID	Temporäre Node-ID
Aktive Node-ID	Aktive Node-ID des Gerätes

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

NMT Status	Persistant Node-ID	Pending Node-ID	Aktive Node-ID
Reset Application			
Reset Communication			
LSS Set Node-ID		Set new value	
LSS Store Node-ID			

Die Aktive 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 Persistant 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 Persistant Node Id muss von der Applikation als Standard Node-ID bereitgestellt werden. Wenn die Persistant 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 (=> Persistant 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.

8.15 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 Concise-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 coEventRegister_CFG_MANAGER()) angezeigt.

Die Konfiguration der Slaves erfolgt über SDO Transfers. Daher muss für jeden zu konfigurierenden Knoten der entsprechende ein SDO Client eingerichtet sein. Für den Knoten 32 zum Beispiel SDO Client 32. Die parallele Konfiguration von mehreren Slaves ist möglich.

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

8.16 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.

8.17 Kommunikations-Status Auswertung

Kommunikationsstatus-Übergänge können durch die Hardware getriggert (Bus-Off, Error Passiv, Overflow, Nachrichtenempfang, Sende-Interrupt)), oder durch einen Timer ausgelöst werden (Return vom Bus-Off). Diese werden über die registrierte Event Funktion (siehe coEventRegister_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 (Kommunikationssta	Beschreibung

	tus)	
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.

8.18 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 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	Sleep-Mode 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 Communikation und dem Versenden der Wakeup-Nachricht.

Mit der Funktion *coSleepModeActive()* kann geprüft werden, ob einer Phase der Sleep-Modes aktiv sind.

8.19 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 coEventRegister_MANAGER_BOOTUP() 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

9 Timer Handling

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

Ein neues Timer-Ereignis 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 Timer-Ticks nur der erste Timer-Vorgang geprüft werden, da alle weiteren Timer noch nicht abgelaufen sein können.

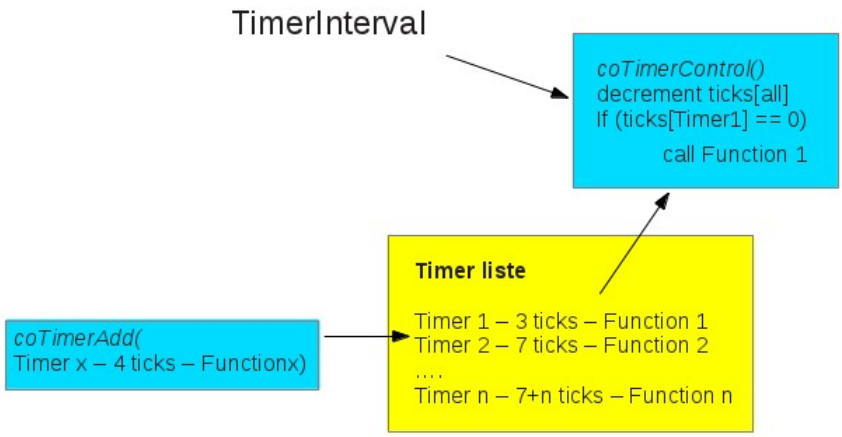


Abbildung 12: Timer Handling

Die notwendigen Timer-Strukturen 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 Timer-Ticks sein werden, wird die angegebene Zeit gerundet. In welche Richtung (auf- oder abrunden) dies geschieht, kann bei der Funktion *coTimerStart()* als Parameter übergeben werden.

10 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.

Die Timerperiode sollte so gewählt werden, dass der *coCommTask()* Aufruf mindestens 2-mal darin erfolgen kann.

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.

10.1 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.

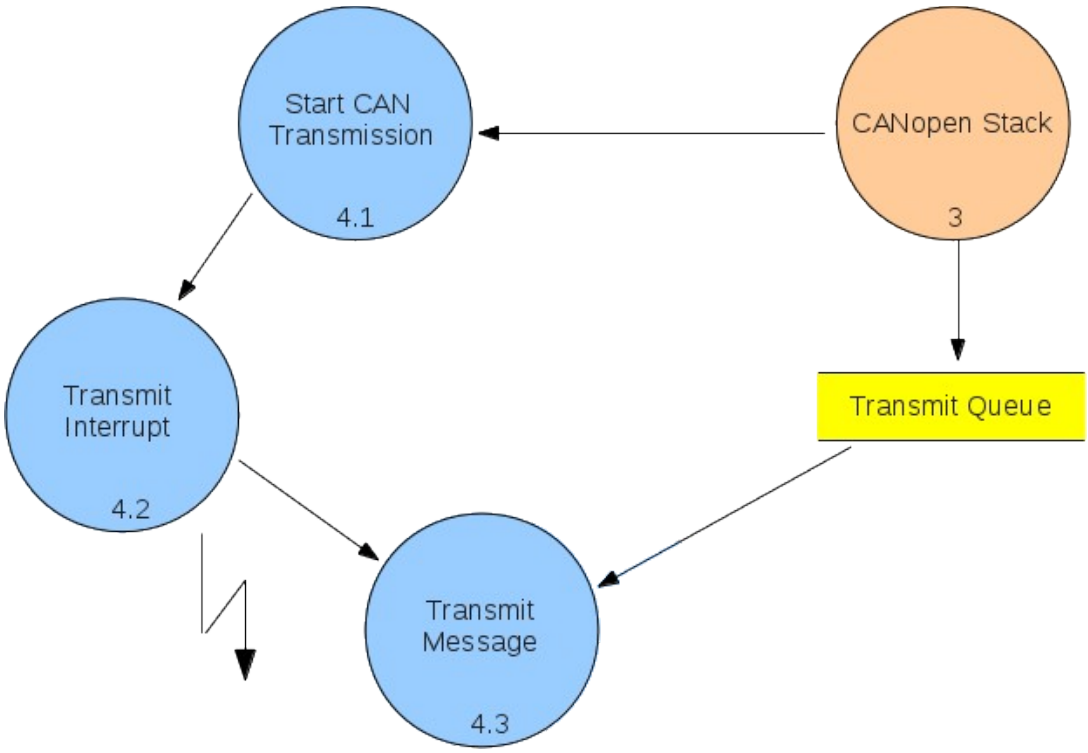


Abbildung 13: CAN Transmit

10.2 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.

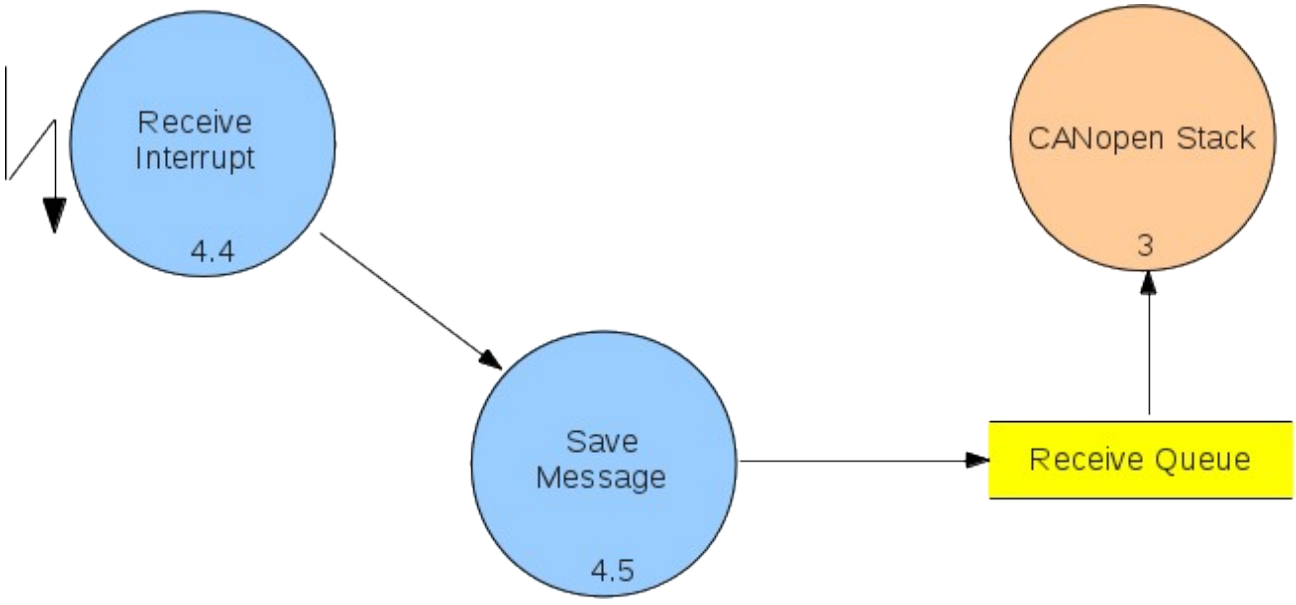


Abbildung 14: CAN Receive

10.3 User-spezischer CAN-Treiber - Interface Beschreibung

Das Treiber Interface verbindet den CANopen Stack mit dem CAN Treiber über verschiedene Funktionsaufrufe, um das Senden bzw. Empfangen von CAN Nachrichten zu ermöglichen. Die Einbindung eines eigenen Treibers kann damit erfolgen.

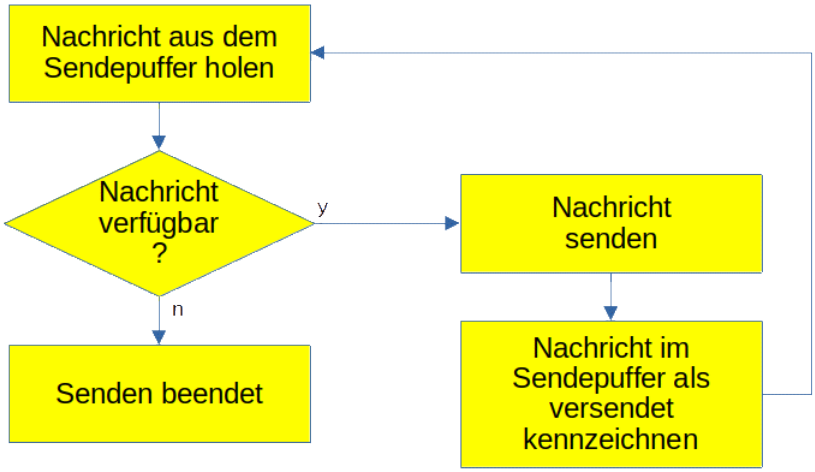
10.3.1 Initialisierung

Die Initialisierung des CAN Treibers erfolgt über die Funktion *codrvCanInit()* mit der gewünschten CAN-Bitrate. Der CAN Controller befindet sich am Ende der Funktion im Zustand "*Disabled*", und muss noch mit der Funktion *codrvCanEnable()* in den Arbeitszustand "Bus On" geschaltet werden, wobei auch notwendige Interrupts aktiviert werden.

10.3.2 CAN-Nachrichten senden

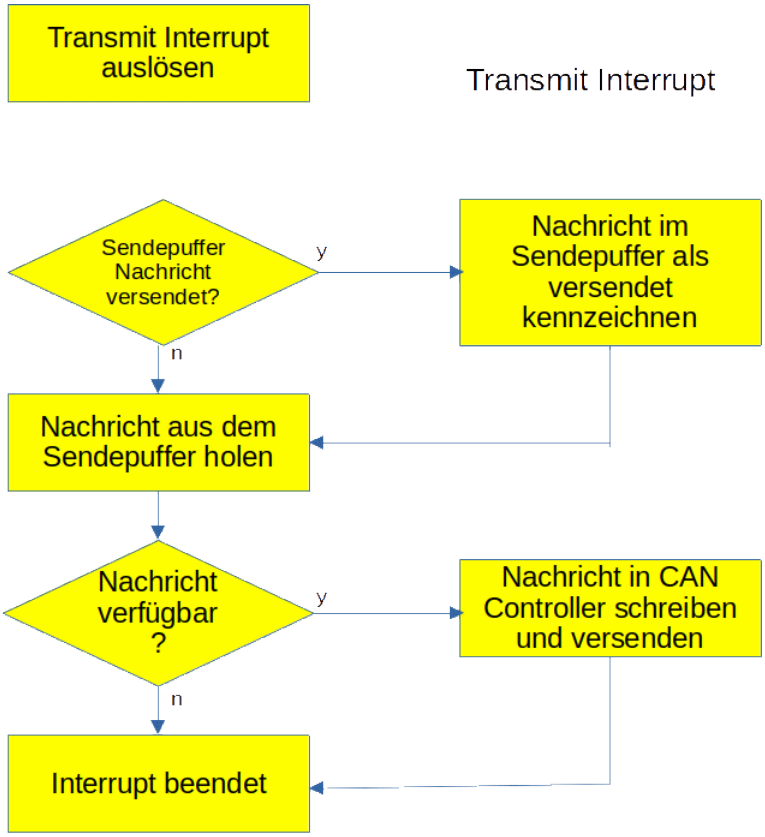
Das Senden von CAN-Nachrichten wir mit der Funktion *codrvCanStartTransmission()* gestartet. Hiermit werden alle Nachrichten aus dem Sendepuffer geholt, an den CAN Controller übergeben und nach erfolgreichem Senden die Nachrichten im Sendepuffer als versendet gekennzeichnet. Die dafür notwendigen Funktionsaufrufe unterscheiden sich für Device-Driver mit Betriebssystem oder reinen embedded Systemen.

Anbindung mit Betriebssystem oder vorhandenen CAN-Treiber



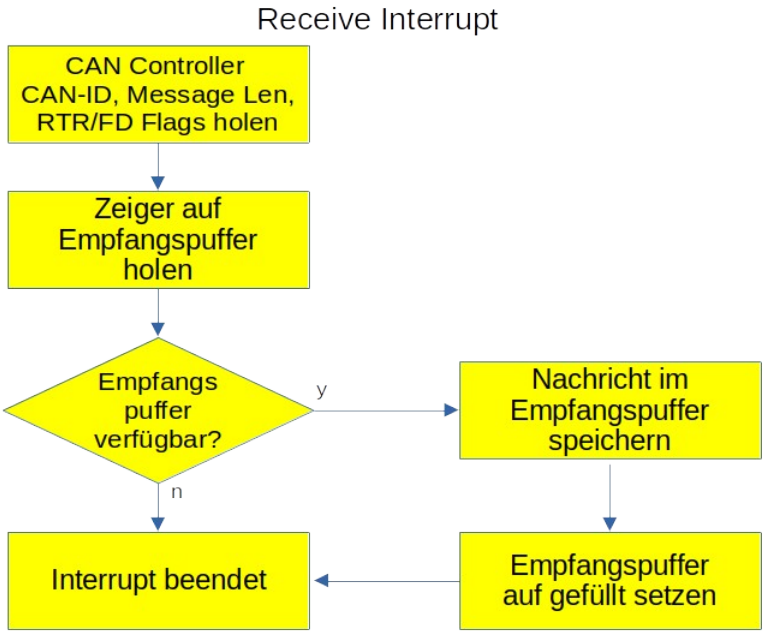
Anbindung mit embedded Treiber

Hier wird der Vorgang mit dem Auslösen eines Sendeinterrupts gestartet.



10.3.3 CAN-Nachrichten empfangen

Empfangene CAN Nachrichten müssen in die Empfangsqueue des Stacks eingetragen werden. Dies kann direkt im Empfangsinterrupt, aber auch außerhalb erfolgen.



10.3.4 CAN Status melden

Bei CAN Controller Status Änderungen (Bus Off, Error Passive, Error Aktive) kann der aktuelle Zustand an den Stack über die Funktion *CO_COMM_STATE_EVENT()* gemeldet werden, so dass anschließend die entsprechenden Indikation Funktionen aufgerufen werden.

Um CAN Status Informationen im Interrupt zu bearbeiten, sollte die Funktion *codrvCanDriverHandler*() genutzt werden.

11 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

11.1 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
- Timer-interrupt

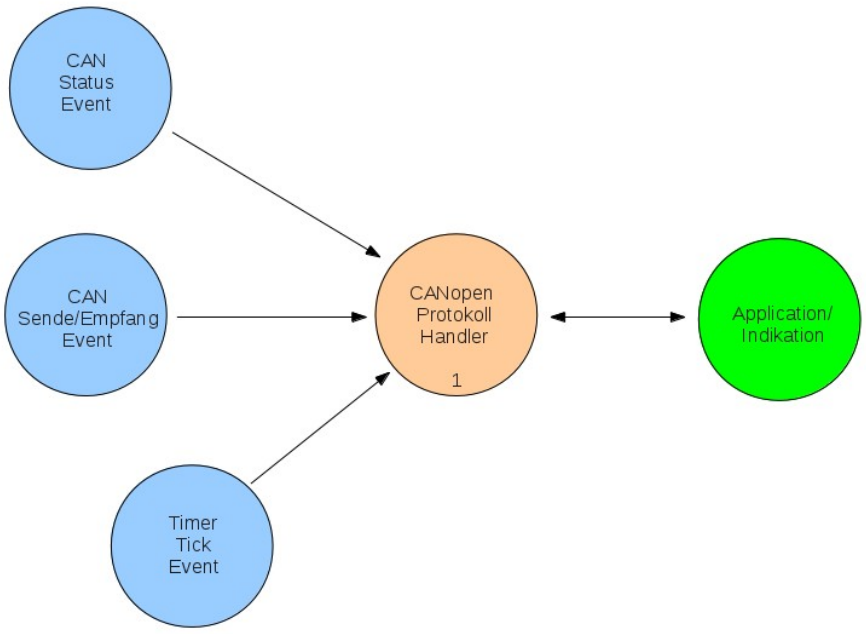


Abbildung 15: Prozess Signal Handling

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

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

11.2 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.

11.3 Mailbox-API

Die Mailbox-API ist ein Konzept, dass inter task Kommunikation nutzt, um den CANopen-Stacks in einem Betriebssystem zu nutzen. Der CANopen-Stack läuft dabei in einem eigenen Thread/Task. Von einer beliebigen Anzahl von Applikations-threads[2] können Kommandos an den CANopen-Thread über Messagequeues gesendet werden. Das Senden von Kommandos entspricht damit dem Funktionsaufruf einer herkömmlichen API-Funktion.

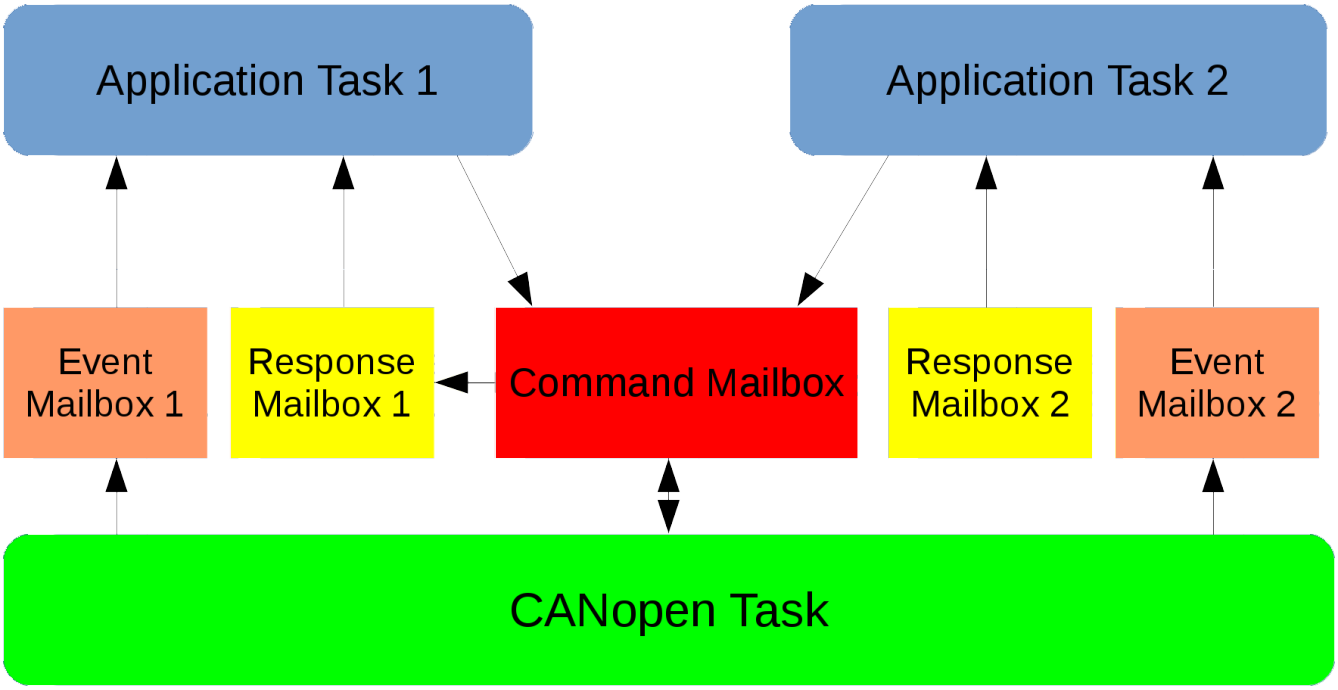


Abbildung 16: Mailbox-API

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 Applikations-thread informiert werden soll. Dieses Berlinverhandlung 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.

2 Nachfolgend wird der Begriff Thread verwendet. Ob es tatsächlich Threads oder Tasks sind hängt vom verwendeten Betriebssystem ab.

11.3.1 Einrichtung eines Applikations-threads

Jeder Applikations-thread 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);
}

/* 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 Applikations-thread 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.

11.3.2 Senden von Kommandos

Für alle grundlegende CANopen-Funktionen und wichtige CANopen-Master-Funktionen[3] 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);
       *-------------------------------------------------------------*/
```

Der Rückgabewert von requestCommand() ist dabei eine selbstständig hoch-laufende 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 */
    Mbx_WaitForResponseMbx(mqResp, &response, 1);      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);
```

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.

3 Weitere CANopen-Funktionen können bei Bedarf als Kommando implementieren werden.

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

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

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

11.3.3 Empfang von Events

Nachdem der Empfang von CANopen-Events durch den Applikations-Thread 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 Indikation-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;
                }
}
```

12 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 zu finden.

13 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.

13.1 SDO Networking

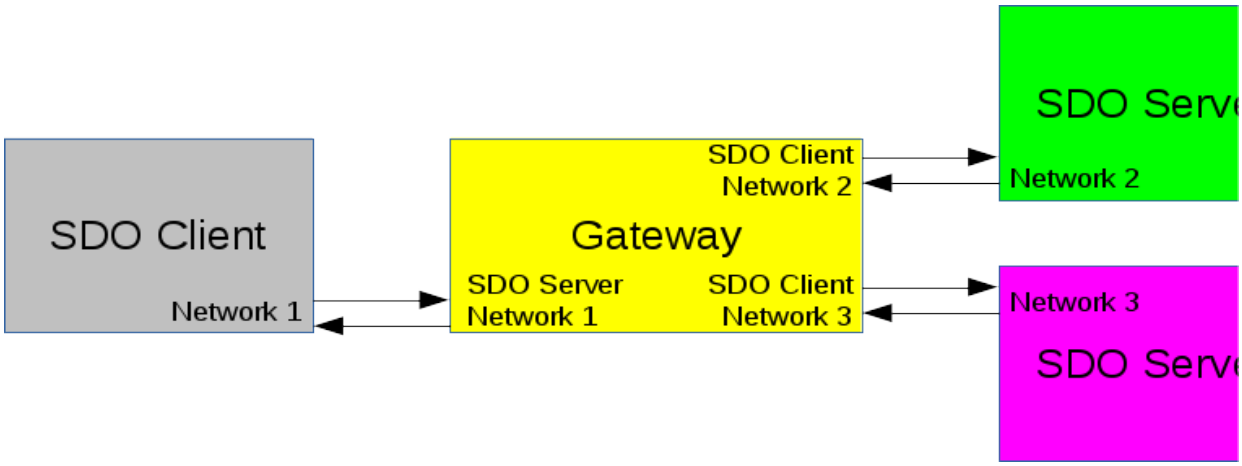


Abbildung 17: Multi-Level Networking

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.

13.2 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.

13.3 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.

14 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 example_sl/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_sl/src (CANopen Stack)

   - Files unter colib_sl/inc (CANopen Stack interne Header)

   - Files unter example_sl/slave1 (Applikation)

   - Files unter codrv_sl/<drivername> (Treiber)
4. Include Pfade setzen

   - example_sl/slave1

   - colib_sl/inc

   - codrv_sl/<drivername>
5. Projekt übersetzen

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

Bitte beachten Sie für eigene Projekte, dass in ihren eigenen Sourcen, in die der Canopen Stack eingebunden wird, immer *gen_define.h* vor *co_canopen.h* eingebunden werden muss.

Files im Beispielprojekt slave1

gen_define.h	Generiertes Files vom CANopen Device Designer, enthält Konfiguration für den Stack
gen_objdict.c	Generiertes Files vom CANopen Device Designer, enthält Objektverzeichnis und Initialiserungsfunktion
main.c	Hauptprogramm
Makefile	Makefile
lave1.cddp	Konfigurationsfile für CANopen DeviceDesigner
slave1.eds	EDS File, generiert durch CANopen DeviceDesigner

14.1 Anpassungen für Hardware und Entwicklungsumgebung

Alle hardware-nahen Files befinden sich im Verzeichnis codrv_sl/<drivername>. Dazu zählen die Bedienung des CAN-Controllers als auch die Anbindung an die Entwicklungsumgebung bzw. Hardwaredefinitionen.

Die Anpassung an die Entwicklungsumgebung bzw. Hardware befinden sich im File cpu_<prozessor.c bzw. cpu_<prozessor>.h. Diese beiden Files müssen an die eigene Anforderungen angepasst werden.

Kopieren sie dafür die beiden Files an eine beliebige Stelle in ihrem Projekt unter einem anderen Namen und nutzen sie diese anstatt der orginalen Files. Damit ist sichergestellt, dass die orginalen Files nicht nach einem Updaten versehentlich überschreiben werden.

15 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.

16 Dienste Schritt für Schritt

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

16.1 SDO Server Nutzung

Einrichtung im CANopen DeviceDesigner:

- Je Server-SDO ein Objekt im Bereich 0x1200 bis 0x127F anlegen
  (Hinweis: COB-IDs dürfen auf Grund des Pre-defined Connection Set nicht gesetzt werden - dies muss im Programm erfolgen)
- Bei Blocktransfer die Parameter:
  - Blöckgröße 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

16.2 SDO Client Nutzung

Einrichtung im CANopen DeviceDesigner:

- Je Client SDO ein Objekt im Bereich 0x1280 bis 0x12FF anlegen
  (Hinweis: COB-IDs dürfen auf Grund des Pre-defined Connection Set nicht gesetzt werden - dies muss im Programm erfolgen)
- Bei Blocktransfer die Paramter:
  - Blöckgröße 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(), coSdoDomanWrite())
- 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

16.3 USDO Server Nutzung

Einrichtung im CANopen DeviceDesigner:

- *Anzahl der Indikation Funktionen eintragen*

Einrichtung in der Applikation:

- Indikation Funktion für Lesen/Schreiben/Test einrichten (coEventRegister_USDO_SERVER_READ() / coEventRegister_USDO_SERVER_WRITE() / coEventRegister_USDO_SERVER_CHECK_WRITE())

Nutzung in der Applikation:

- erfolgt asynchron beim Eintreffen von USDO-Anfragen

  der Rückgabewert hat Einfluß auf die Antwort des USDO Transfers

16.4 USDO Client Nutzung

Einrichtung im CANopen DeviceDesigner:

- *Anzahl der Indication eintragen*

Einrichtung in der Applikation:

- Indikation Funktion für das Ergebnis auf Lesen/Schreiben Aufruf einrichten (coEventRegister_USDO_CLIENT_READ() / coEventRegister_USDO_CLIENT_WRITE())

Nutzung in der Applikation:

- Transfer starten (coUSdoRead(), coUSdoWrite(), coUSdoDomanWrite())
- Das Ergebnis des Transfers wird über die eingerichtete Indikation Funktion geliefert
- Bei Nutzung des Domaintransfers (coUSdoDomainWrite()) kann zusätzlich eine Indikation Funktion angegeben werden, die nach einer definierten Anzahl von Nachrichten aufgerufen wird, um z.B. den Domainpuffer zu aktualisieren

16.5 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

16.6 Emergency Producer

16.6.1 CANopen classic

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

16.6.2 CANopen FD

Einrichtung im CANopen DeviceDesigner:

- Emergency Producer Objekt (0x1014) anlegen
- Error History Objekte 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

16.7 Emergency Consumer

Einrichtung im CANopen DeviceDesigner:

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

Einrichtung in der Applikation:

- Indikation Funktion für den Empfang der Emergency Nachricht einrichten (coEventRegister_EMCY_CONSUMER())

Nutzung in der Applikation:

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

16.8 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

16.9 PDOs

16.9.1 Empfangs-PDOs

Einrichtung im CANopen DeviceDesigner:

- Objekte, die mit PDO empfangen 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 (Objekte 0x1400.. 0x15ff) anlegen:
  - Transmission Type festlegen – bei synchronen PDOs müssen auch die SYNC Objekte (siehe 16.8) 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.

16.9.2 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 16.8) 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 Syncronen PDOs (Transmission Type 1..240)
    - versenden erfolgt automatisch

16.10 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.

16.11 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.

16.12 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_CFG_MANAGER) 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.

17 Aufbau der Verzeichnisstruktur

codrv_sl/xxx	Hardwarespezifische CANopen SingleLine Treiber
codrv_sl/common	Allgemeine CANopen SingleLine Treiber Dateien
colib_sl/inc	CANopen SingleLine Protokoll Stack Header
colib_sl/src	CANopen SingleLine Protokoll Stack Sourcen und interne Header
colib_sl/profile	CANopen SingleLine Profile
colib_sl/csharp_wrapper	CANopen SingleLine C# Wrapper
example_sl	CANopen SingleLine Beispiele
codrv_ml/xxx	Hardwarespezifische CANopen MultiLine Treiber
codrv_ml/common	Allgemeine CANopen MultiLine Treiber Dateien
colib_ml/inc	CANopen MultiLine Protokoll Stack Header
colib_ml/src	CANopen MultiLine Protokoll Stack Sourcen und interne Header
colib_ml/profile	CANopen MultiLine Profile
example_ml	CANopen MultiLine Beispiele
cofddrv_sl/xxx	Hardwarespezifische CANopenFD SingleLine Treiber
cofddrv_sl/common	Allgemeine CANopenFD SingleLine Treiber Dateien
cofdlib_sl/inc	CANopenFD SingleLine SingleLine Protokoll Stack Header
cofdlib_sl/src	CANopenFD SingleLine Protokoll Stack Sourcen und interne Header
examplefd_sl	CANopenFD SingleLine Beispiele
cofddrv_ml/xxx	Hardwarespezifische CANopenFD MultiLine Treiber
cofddrv_ml/common	Allgemeine CANopenFD MultiLine Treiber Dateien
cofdlib_ml/inc	CANopenFD MultiLine Protokoll Stack Header
cofdlib_ml/src	CANopenFD MultiLine Protokoll Stack Sourcen und interne Header
examplefd_ml	CANopenFD MultiLine Beispiele
ref_man	Referenz Manual
user_man	User Manual

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