Dokumentation für Anwender:
CoDeSys V 2.3
CANopen für 3S Laufzeitsysteme

Dokument Version 1.4

tech_doc_d.dot / V1.2

INHALT

1 VORBEMERKUNG 4

1.1 Allgemein 4

1.2 CANopen-Begriffe und 3S Implementation 4

1.3 Aufbau des Dokuments 5

2 CANOPENMASTER 6

2.1 Abgrenzung zu anderen CANopen Bibliotheken 6

2.2 CANopen für den Anwender 7

2.2.1 Verwendete Module 7

2.2.2 Vorgehen bei der Erstellung eines Projekts mit CANopen 7

2.2.2.1 Erstellung eines neuen Projekts und Zielsystemauswahl 7

2.2.2.2 Steuerungsapplikation erstellen 8

2.2.2.3 Einbindung der erforderlichen Bibliotheken 8

2.3 Funktionalität der Bibliotheken zur Laufzeit 8

2.3.1 Implizite Dienste der CANopenMaster Bibliothek 8

2.3.2 Explizite Dienste der CANopen Bibliothek 9

2.3.3 Starten des CANopen-Netzwerks 9

2.4 Arbeiten von Applikationsseite mit dem CANopen-Master-Teil 10

2.4.1 Die Struktur des CANopen-Masters 10

2.4.2 Die Struktur des CANopen-Nodes (Slaves) 13

2.4.3 Zustände der Bibliothek interpretieren und reagieren 15

2.4.3.1 Hochlauf des CANopen-Masters 15

2.4.3.2 Hochlauf der CANopen-Slaves 16

2.4.3.3 Nodeguarding-/Heartbeatfehler 17

2.4.3.4 Node antwortet nicht in Konfigurationsphase 17

2.4.3.5 Antworten der Slaves auf die Konfigurations-SDOs auswerten 17

2.4.4 Beispiele für Applikationsgesteuerte Aktionen der Bibliotheken 17

2.4.4.1 SDO-Transfer über den Funktionsblock „CanOpenSendSDO“ 18

2.4.4.2 Hochlauf des Netzwerks ohne automatisches Starten 21

2.4.4.3 Stoppen eines Nodes 22

2.4.4.4 Zurücksetzen eines Nodes 22

2.4.4.5 Starten des Netzwerks mit bUseStartAllNodes 22

2.4.4.6 Initialisieren des Netzwerks mit bUseResetAllNodes 23

2.4.4.7 Überwachen der Konfigurationsphase 23

2.4.4.7.1 Zugriff auf den Status des CANopen-Masters 23

2.4.4.7.2 Zeitüberwachung der Hochlaufphase 23

tech_doc_d.dot / V1.2

2.5 Implizite Aufrufe 24
2.6 Das Objektverzeichnis des CANopen-Masters 24
3 CANDEVICE 26
3.1 Funktionalität 26
3.2 CanDevice konfigurieren 26
3.3 CanDevice Einstellungen 27
3.4 EDS-Datei erzeugen 29
3.5 Verändern des Standard-Mappings durch Master-Konfiguration 32
3.6 Arbeiten vom Anwenderprogramm mit dem CanDevice 33
3.6.1 Der Baustein CanopenDevice 33
3.6.2 Zugriff auf die OD-Einträge vom Anwenderprogramm 35
3.6.3 Ändern der PDO-Eigenschaften zur Laufzeit 36
3.6.4 Emergency-Nachrichten durch das Anwenderprogramm senden 36
4 CAN NETZWERKVARIABLEN 38
4.1 CAN-Netzwerk-Variablen für den Anwender 38
4.1.1 Voraussetzungen 38
4.1.2 Einstellungen in den Zielsystemeinstellungen 38
4.1.3 Einstellungen in den globalen Variablenlisten 38
4.1.4 Erzeugte Aufrufe 40
5 ANHANG 42
5.1 Liste der normalerweise für einen Slave erzeugten SDOs 42
5.2 Erkennen eines Schreibpufferüberlaufs 42
5.3 FAQs – Häufig gestellte Fragen 42
5.3.1 CANopen Funktionalität 42
5.3.2 CANopen Master 46
5.3.2.1 Konfiguration 46
5.3.3 Applikation 47
5.3.4 Mögliche Fehler, Ursachen und deren Behebung 47
ÄNDERUNGSHISTORIE 49

tech_doc_d.dot / V1.2

1 Vorbemerkung

1.1 Allgemein

CoDeSys ist eines der führenden Systeme für die Programmierung von Steuerungssystemen nach dem internationalem Standard IEC 61131. Um CoDeSys für den Anwender und für den OEM-Kunden interessanter zu gestalten, wurden viele wichtige Funktionen in das Programmiersystem integriert. Darunter auch ein Konfigurator für CANopen. Mit diesem CANopen-Konfigurator ist es möglich CANopen Netzwerke unter CoDeSys zu konfigurieren. Die mit CoDeSys programmierte Steuerung dient in diesem Netzwerk als CANopen-Master.

Bis zum Mitte des Jahres 2000 wurden die Daten des Konfigurationsvorgangs von CoDeSys nach dem Programm-Download zum Zielsystem übertragen.
Auf dem Zielsystem musste ein Software-Modul diese Daten entgegennehmen, interpretieren, verwalten und verarbeiten. Die eigentliche CANopen Bearbeitung konnte durch das von 3S - Smart Software Solutions gelieferte Kernlaufzeitsystem nicht geleistet werden. Als Folge wurde von vielen Kunden eine CANopen Protokoll-Software in das Laufzeitsystem integriert.

Um die Nachteile dieses Ansatzes zu eliminieren, wurde von 3S – Smart Software Solutions eine eigene, **eingeschränkte** CANopen Implementation realisiert. Diese ist, damit sie einem breiten Anwenderkreis zugänglich gemacht werden kann, als CoDeSys Bibliothek in IEC 61131-3 implementiert. Die Bibliothek stützt sich auf sehr einfache Basis-CAN-Funktionen ab, die als CAN-Treiber bezeichnet werden.

Durch die Realisierung der CANopen-Funktionen als CoDeSys Bibliothek ist eine einfache Skalierung des Zielsystems möglich. So verbraucht die CANopen-Funktion nur dann Zielsystem-Ressourcen, wenn die Funktion auch wirklich genutzt wird. Zur weiteren Schonung von Zielsystem Ressourcen wird durch CoDeSys automatisch eine genau der Konfiguration entsprechende Datenbasis für die CANopen-Master Funktion generiert. Weiterhin muss das Update der Ein- und Ausgaben auf dem Zielsystem nicht mehr interpretiert werden, sondern wird durch automatisch generierten Code realisiert.

Ab der Programmiersystemversion 2.3 kann eine CoDeSys-Steuerung nun auch ein CANopen-Slave (CAN-Device) sein. Dazu wurde ein neues Unterelement, das über geeignete Beschreibungsdateien der Konfiguration (.cfg-Datei) sichtbar wird, in den Konfigurator integriert.

Die hier beschriebenen Bibliotheken können so ab der CoDeSys Version 2.3.5.0 benutzt werden.

1.2 CANopen-Begriffe und 3S Implementation

Nach CANopen gibt es keine Master und Slaves an einem CAN-Netz.

Es gibt nach CANopen einen NMT-Master, einen Konfigurationsmaster usw., immer mit der Vorstellung, dass alle Teilnehmer eines CAN-Netzes gleichberechtigt sind.

Die Implementation von 3S geht davon aus, dass ein CAN-Netz als Peripherie einer CoDeSys-programmierbaren Steuerung dient. Demzufolge wird eine SPS im CAN-Konfigurator von CoDeSys als CAN-Master bezeichnet. Dieser Master ist NMT-Master und Konfigurationsmaster. Im Normalfall wird die SPS dafür sorgen, dass das Netz in Betrieb genommen werden kann. Der Master übernimmt die Initiative, die einzelnen Knoten (Nodes), die ihm per Konfiguration bekannt sind, zu starten. Diese Knoten werden als „Slaves“ bezeichnet.

Um den Master ebenfalls dem Status eines CANopen-Nodes näher zu bringen, wurde ein Objektverzeichnis für den Master eingeführt. Auch kann der Master als SDO-Server auftreten, nicht mehr nur in der Konfigurationsphase der Slaves als SDO-Client. (Die Erklärung folgt in einem späteren Kapitel.)

tech_doc_d.dot / V1.2

1.3 Aufbau des Dokuments

Dieses Dokument ist grob in 3 Teile aufgeteilt: im ersten Teil wird die CANopen-Master-Bibliothek, im zweiten die CANopen-Slave-Bibliothek beschrieben. Den CAN Netzwerkvariablen ist ebenso ein eigenes Kapitel gewidmet.

Die Beschreibungen sind jeweils vor allem in folgende Themen gegliedert:

- Übersicht über die Leistung der Bibliothek.
- Beschreibung des Standardvorgehens bei der Arbeit mit der jeweiligen Bibliothek (im folgenden auch kurz „Lib“ genannt).
- Beschreibung der Möglichkeiten, die die Applikation hat, auf den CAN-Stack Einfluss zu nehmen und Information von diesem zu erhalten.
- Eine Übersicht über mögliche Fehlerzustände und deren Ursachen.

© 3S - Smart Software Solutions GmbH

2 CANopenMaster

2.1 Abgrenzung zu anderen CANopen Bibliotheken

Die von 3S - Smart Software Solutions realisierte CANopen Bibliothek grenzt sich in verschiedenen Punkten von auf dem Markt befindlichen Systemen ab. Sie wurde nicht entwickelt, um andere Bibliotheken namhafter Hersteller überflüssig zu machen, sondern ist bewusst für den Einsatz mit CoDeSys Programmier- und Laufzeitsystem optimiert.

Die Bibliotheken sind nach der Spezifikation der CiA DS301, V401 gemacht.

Für den Anwender der CoDeSys CANopen Bibliothek ergeben sich folgende Vorteile:

- Die Implementation ist unabhängig vom Zielsystem und damit praktisch auf jeder CoDeSys-programmierbaren Steuerung direkt verwendbar.
- Der Kunde erhält ein komplettes System aus einer Hand.
  Das bedeutet, der CANopen Konfigurator, die Einbindung in das Entwicklungs- und in das Laufzeitsystem sind optimal aufeinander abgestimmt.
- Die CANopen Funktionalität ist nachladbar.
  Das bedeutet, dass die CANopen-Funktionen ohne Änderung des Laufzeitsystems geladen, entladen und aktualisiert werden können.
- Die Ressourcen des Zielsystems werden geschont.
  Dies wird durch die automatische konfigurationsabhängige Speicherbelegung erreicht. Es werden also nicht die Ressourcen für eine Maximalkonfiguration vorbehalten.
- Automatisches Aktualisieren der Ein- und Ausgänge ohne zusätzliche Maßnahmen.

Folgende in CANopen definierten Funktionen werden von der 3S – Smart Software Solutions CANopen Bibliothek unterstützt (Stand: Mai 2005):

- PDOs Senden vom Master => zu den Slaves
  ereignisgesteuert (d.h. bei Änderung), zeitgesteuert (RepeatTimer) und als synchrone PDOs, d.h. immer nach einer bestimmten Anzahl von SYNC Nachrichten auf dem Bus (die Anzahl wird vom Transmission Type des PDOs im Can-Konfigurator vorgegeben) wird das PDO vom Master gesendet.. Auch eine externe Sync-Quelle kann benutzt werden, um das senden von synchronen PDOs zu initiieren.
- PDO Empfangen vom Slave => zum Master
  je nach Slave: ereignisgesteuert, abfragegesteuert, azyklisch und zyklisch
- PDO Mapping
  wenn vom Slave unterstützt
- SDO Senden und Empfangen (unsegmentiert, d.h. 4 Bytes pro Objektverzeichniseintrag) automatische Konfiguration aller Slaves über SDOs beim Systemstart applikationsgesteuertes Senden und Empfangen von SDOs zu konfigurierten Slaves
- Synchronisation
  automatisches Senden von Sync-Nachrichten durch den CANopen Master
- Nodeguarding
  automatisches Senden von Guarding-Nachrichts und Überwachung der Lifetime für jeden entsprechend konfigurierten Slave
- Heartbeat
  automatisches Senden und Überwachen von Heartbeat-Nachrichten
- Emergency
  Empfangen und Speichern von Emergency-Nachrichten von den konfigurierten Slaves

tech_doc_d.dot / V1.2

- Die direkte Kommunikation von Slaves über PDOs ist möglich, muss jedoch „von Hand“ konfiguriert werden.

Folgende in CANopen definierten Funktionen werden von der 3S – Smart Software Solutions CANopen Bibliothek derzeit nicht unterstützt (Stand: Mai 2005):

- Dynamische Identifier-Zuordnung
- Dynamische SDO-Verbindungen
- Blockweiser SDO-Transfer
- Die Überwachung mehr als eines Heartbeats vom CAN Device aus.
- Netzwerkvariablen im Sinne der CANopen Spezifikation DS302
- Bausteine im Sinne der DS405.
- Alle oben nicht genannten Möglichkeiten des CANopen Protokolls.

2.2 CANopen für den Anwender

2.2.1 Verwendete Module

Zur einfachen und praxisnahen Realisierung der CANopen Funktion unter einer CoDeSys programmierbaren Steuerung sind folgende Module erforderlich:

- Ein CoDeSys Programmiersystem ab Version 2.3 SP5, mit integriertem CANopen Konfigurator
- Die STANDARD.LIB, welche die in der IEC-61131 definierten Standardfunktionen für die Steuerung zu Verfügung stellt.
- In CoDeSys-programmierbaren Multitaskingsystemen werden die SysLibSem (Semaphoren zur exklusiven Ausführung bestimmter Codestellen) und SysLibCallback (IEC-Funktionsaufrufe bei bestimmten Ereignissen) benutzt, um den Zugriff auf den CAN multitaskingsicher zu machen.
- EDS – Dateien für alle Slaves, die am Netzwerk betrieben werden sollen. Die EDS-Dateien werden vom Hardware-Lieferanten der CANopen Slaves zu Verfügung gestellt.
- Die 3S_CANDRV.LIB, welche die CAN Basisfunktionen realisiert. Die Bibliothek ist hardware-abhängig und muss als externe (oder interne) CoDeSys Bibliothek zur Verfügung stehen. Diese Bibliothek wird in der Regel vom Steuerungshersteller zur Verfügung gestellt, und besitzt eine einheitliche Schnittstelle zur 3S_CANOPEN.lib.
- Die 3S_CanOpenManager.LIB, welche die CANopen Basisfunktionalitäten zur Verfügung stellt.
- Eine oder mehrere der Bibliotheken 3S_CanOpenNetVar, 3S_CanOpenDevice und 3S_CanOpenMaster. Je nach gewünschter Funktionalität.

2.2.2 Vorgehen bei der Erstellung eines Projekts mit CANopen

Die Erstellung eines neuen Projektes mit CANopen wird nachfolgend schrittweise beschrieben. Dabei wird davon ausgegangen, dass CoDeSys auf dem Rechner bereits fertig installiert ist, die Target- und EDS-Dateien ebenfalls richtig installiert bzw. kopiert wurden.

Hinweis: Eine Beschreibung der Dialoge und Menübefehle des CAN-Steuerungskonfigurators ist auch in der CoDeSys Benutzerdokumentation (Benutzerhandbuch, Online Hilfe) enthalten.

2.2.2.1 Erstellung eines neuen Projekts und Zielsystemauswahl

Nach dem Start von CoDeSys wählt man => **Datei | Neu**

Im Dialog ‚Zielsystemeinstellungen’ wählt man das gewünschte Zielsystem aus.

Anschließend aktiviert man im Register **Allgemein** den Punkt
CANopen Konfiguration unterstützen. Wenn dieser Punkt im Dialog nicht sichtbar ist, und trotzdem kein CAN Master in der Steuerungskonfiguration eingefügt werden kann, wird CANopen von diesem Zielsystem nicht unterstützt.

tech_doc_d.dot / V1.2

2.2.2.2 Steuerungsapplikation erstellen

Nach dem Schließen der Zielsystemeinstellungen mit OK öffnet CoDeSys automatisch eine Dialogbox zum Einfügen von PLC_PRG. Nachdem dieser Dialog nach Auswahl der Programmiersprache (ST, KOP, FUP…) mit o.k. verlassen wurde, kann nun eine Applikation programmiert werden.

2.2.2.3 Einbindung der erforderlichen Bibliotheken

Die erforderlichen Bibliotheken 3S_CanOpenManager.LIB und 3S_CANDRV.LIB und eine oder mehrere der Bibliotheken 3S_CanOpenNetVar/Master/Device.lib müssen in das Projekt aufgenommen werden. Hierzu verwendet man: **Fenster | Bibliotheksverwaltung** und anschließend für jede einzelne der Bibliotheken **Einfügen | weitere Bibliothek**, aus dem Kontextmenu des Bibliotheksverwalters oder der Menuleiste.

Hinweis:
Wird eine der Bibliotheken 3S_CanOpenNetVar/Master/Device.lib eingefügt, so werden die anderen, von ihr benutzten Bibliotheken automatisch mit eingefügt.

Das Einbinden der Bibliotheken ist ein wichtiger Schritt. CoDeSys erkennt am Vorhandensein der Bibliotheken in einem Projekt, dass es sich um ein CANopen Projekt handelt, für das spezielle, implizite Programmteile automatisch erzeugt werden.

Den Umgang mit dem CAN-Konfigurator von CoDeSys selbst entnehmen Sie bitte der Hilfe von CoDeSys. Hier wird erklärt, wie Sie einen CANopen-Master und dazu Slaves konfigurieren können.

Sind die CANopen Bibliotheken in das Projekt eingefügt, kann das Projekt erst nachdem auch ein CANopen Master oder –Device in der Steuerungskonfiguration eingefügt wurde, fehlerfrei übersetzt werden.

2.3 Funktionalität der Bibliotheken zur Laufzeit

2.3.1 Implizite Dienste der CANopenMaster Bibliothek

Die CANopenMaster Bibliothek stellt der CoDeSys Applikation implizite Dienst zu Verfügung, die für die meisten Anwendungen ausreichend sind. Diese Dienste werden für den Anwender transparent integriert und stehen in der Applikation ohne zusätzliche Aufrufe zu Verfügung.

Zu diesen Diensten zählen:

- Der Reset aller konfigurierten Slaves am Bus beim Systemstart — Standard
  Um die Slaves zurückzusetzen wird standardmäßig das NMT-Kommando „Reset Remote Node“ benutzt, explizit für jeden Slave einzeln. (NMT steht nach CANopen für Network Managment. Die einzelnen Kommandos sind im Dokument DSP301 beschrieben.)
  Die Erfahrung einiger Anwender hat gezeigt, dass es sinnvoll ist, die Slaves mit einem Reset All Remote Nodes Kommando zurückzusetzen, um die Empfangskapazitäten von wenig leistungsstarken Slave-Controllern nicht zu überlasten. Das ist über die Applikation durch das Setzen des entsprechenden Flags erreichbar.
- Abfrage des Slave-Gerätetyps mittels SDO (Abfrage des Objekts 0x1000) und Vergleich mit der konfigurierten Slave ID.
  Ausgabe eines Fehlerstatus für die Slaves, von denen ein falscher Gerätettyp empfangen wurde. Wird kein Gerätetyp empfangen, wird die Anfrage nach 0,5 Sekunden wiederholt, außer der Slave wurde als optional in der Konfiguration markiert. — Standard
- Konfiguration aller fehlerfrei detektierten Geräte mittels SDO. Jedes SDO wird auf Antwort überwacht und wiederholt, wenn sich innerhalb der Überwachungszeit der Slave nicht meldet. — Standard
- Automatische Konfiguration von Slaves mittels SDO’s bei laufendem Busbetrieb wenn sich der Slave mittels BOOT-Up Nachricht beim Master anmeldet. — Standard

tech_doc_d.dot / V1.2

- Start aller fehlerfrei konfigurierten Slaves nach dem Ende der Konfiguration, wenn die Option „Automatisch Starten“ des Can Masters aktiviert wurde. einstellbar
  Zum Starten der Slaves wird normalerweise das NMT-Kommano „Start remote node“ benutzt. Wie beim Reset kann dieses durch „Start all remote nodes“ ersetzt werden.
- Zyklisches Senden der Sync-Nachricht einstellbar
- Nodeguarding mit Lifetime-Überwachung für jeden Slave einstellbar
  Ausgabe eines Fehlerstatus für die Slaves mit Lifetime-Fehler
- Heartbeat vom Master an die Slaves und Überwachen der Heartbeats der Slaves. einstellbar
- Empfangen von Emergency-Nachrichten für jeden Slave mit Speicherung der zuletzt empfangenen Emergency-Nachrichten für jeden Slave getrennt Standard
- Empfangen von PDO-Nachrichten und task-konsistentes Übertragen der Daten in das Prozessabbild der Applikation Standard
- Senden von PDO-Nachrichten nach dem Ende der Task, je nach eingestelltem „Transmission Type“ des PDOs. Standard

2.3.2 Explizite Dienste der CANopen Bibliothek

Zusätzlich zu den oben beschriebenen impliziten Diensten stellt die CANopen Bibliothek dem Anwender folgende zusätzlichen Funktionen zu Verfügung:

Zu diesen Diensten zählen:

- Anzeige von neu empfangenen PDOs mittels Flag, welches von der Applikation zurückgesetzt werden kann. Standard
- Applikations-getriggertes Senden von Remote Telegrammen für Empfangs-PDOs Standard
- Applikations-getriggertes Senden von SDO Objekten an Slaves Standard
- Applikations-getriggertes Reset von Slaves mit anschließendem Neukonfigurieren von Slaves. Standard
- Die Möglichkeit für die Applikation, eigene CAN-Nachrichten direkt auf den Bus zu legen und direkt zu empfangen besteht ausdrücklich weiterhin. (Layer 2) Standard

2.3.3 Starten des CANopen-Netzwerks

Nach einem Download des Projekts auf die Steuerung oder einem Reset der Applikation wird das CAN-Netz vom Master neu hochgefahren.

Das geschieht immer in der gleichen Reihenfolge von Aktionen:

1. Alle Slaves werden zurückgesetzt, außer wenn sie als „Nicht initialisieren“ im Konfigurator markiert sind. Das Zurücksetzen geschieht einzeln mit dem NMT-Kommando „Reset Node“ (0x81), jeweils mit der NodeId des Slaves. Außer die Applikation hat das Flag „bUseResetAllNodes“ gesetzt, dann wird zum hochfahren des Netzes das Kommando einmal mit NodeID 0 benutzt, „Reset All Nodes“.
2. Alle Slaves werden konfiguriert. Dazu wird zunächst das Objekt 0x1000 (Device-Typ) des Slaves abgefragt. Wenn der Slave innerhalb der Überwachungszeit von 0,5 Sekunden antwortet, wird das jeweils nächste Konfigurations-SDO gesendet. Ist ein Slave als „Optional“ markiert und antwortet nicht innerhalb der Überwachungszeit auf die Abfrage des Objekts 0x1000, wird er als nicht vorhanden markiert und keine weiteren SDOs werden an ihn geschickt. Wenn ein Slave auf die Abfrage des Objekts 0x1000 mit einem anderen als dem konfigurierten Typ in den unteren 16 Bit antwortet, wird er zwar konfiguriert aber als falscher Typ markiert.
3. Alle SDOs (auch die Abfrage des Objekts 0x1000) werden jeweils solange wiederholt, bis innerhalb der Überwachungszeit eine Antwort des Slaves gesehen wurde. Hier kann die Applikation den Hochlauf der einzelnen Slaves überwachen und ggf. reagieren. (siehe dazu unten)

tech_doc_d.dot / V1.2

4. Wenn der Master eine Heartbeat-Zeit ungleich 0 konfiguriert hat, beginnt die Erzeugung des Heartbeats sofort nach dem Starten der Mastersteuerung.
5. Nachdem alle Slaves ihre Konfigurations-SDOs erhalten haben, beginnt für Slaves mit konfiguriertem Nodeguarding das Guarding.
6. Wenn der Master auf automatisch Starten konfiguriert wurde, werden jetzt alle Slaves einzeln vom Master gestartet. Dazu wird das NMT-Kommando „Start Remote Node“ (0x1) benutzt. Wenn die Applikation das Flag „bUseStartAllNodes“ setzt, wird hier das Kommando mit NodeId 0 benutzt und somit alle Slaves mit einem „Start all nodes“ gestartet.
7. Es werden mindestens einmal alle konfigurierten Tx-PDOs (für die Slaves sind das Rx-PDOs) gesendet. (Das Senden von RTRs für RTR-PDOs ist immer Sache der Applikation. Hierfür gibt es eine Methode der Rx-PDOs.)

2.4 Arbeiten von Applikationsseite mit dem CANopen-Master-Teil

Es gibt verschiedene Fälle, in denen die IEC-Applikation mit der CANopenLib zusammenarbeiten muss. Dazu gehören z.B. das Erkennen und Reagieren auf einen Fehler.

Dabei beachten Sie die Aufrufreihenfolge, die vom Applikationsprogrammierer nicht beeinflusst werden kann. (Diese Aufrufe und in welchen Tasks die Aufrufe erzeugt werden sind auch im Kapitel 2.5, Implizite Aufrufe, beschrieben.)

Hier ein Beispiel für einen CAN-Controller und je ein Rx/Tx-PDO:

- CanRead(0); (*Implizit erzeugter Aufruf*)
- pCanOpenMaster[0](); (*Implizit erzeugter Aufruf*)
- pCanOpenPDO_Rx[0](); (*Implizit erzeugter Aufruf*)
- <Abarbeitung des Applikationscodes>
- pCanOpenPDO_Tx[0](); (*Implizit erzeugter Aufruf*)
- MgrClearRxBuffer(wCurTask:= 1,wDrvNr := 0, dwFlags := 0, dwPara := 0); (*Implizit erzeugter Aufruf zum Aufräumen des Empfangsbuffers.*)

2.4.1 Die Struktur des CANopen-Masters

CoDeSys erzeugt beim Vorhandensein der Bibliotheken 3S_CanOpenMaster/3S_CanOpenManager und 3S_CanDrv.lib in einem Projekt implizit (d.h. automatisch) eine globale Variablenliste und Vor/Nach dem Applikationscode in bestimmten Tasks Aufrufe von Bausteinen der Bibliotheken. Die globale Variablenliste hat den Namen „CanOpen implicit Variables“ und wird von CoDeSys mit den entsprechenden Daten aus der Steuerungskonfiguration gefüllt.

Hinweis: Um sich den Initialisierungscode in ST anzusehen, ohne Online gehen zu müssen, kann CoDeSys mit der Kommandozeilenoption /debug gestartet werden. Dann werden (neben anderen nützlichen Informationen) im Übersetzungsverzeichnis die Dateien CanOpenInitcode.exp und CanOpenBeforeTask/AfterTask_<Taskname>.exp erzeugt.

Die Variablenliste „CanOpen implicit Variables“ im Detail:

Die Konstanten geben immer die Grenzen der Arrays an, die in dieser Variablenliste deklariert sind.

```
VAR_GLOBAL CONSTANT
    MAX_CTRLINDEX : INT := <Maximaler Index der CAN-Controller in der CANopen-
                           Konfiguration, bei einem Controller 0, bei 2 Controllern 1 usw.>;
END_VAR
VAR_GLOBAL CONSTANT
    USE_CANOPEN_NODES : BOOL := <Gibt an, ob mindestens 1 Slave unter einem Master
                                 vorhanden ist>;
    MAX_MASTERINDEX : INT :=    <Gibt an, bis zu welchem Index in der Tabelle von
                                 Controllern (in der Globalen Variablenliste der Bibliothek
                                 CanOpenManager) die Controller einem Master und
```

tech_doc_d.dot / V1.2

```
                                        nicht einem CanDevice (siehe Kapitel 3, CanDevice)
                                        zugeordnet sind.>;
    MAX_NODEINDEX : INT :=              <Gibt die Größe des Arrays von CANopen-Nodes an.>;
    MAX_SDOINDEX : INT :=               <Gibt die Größe des Arrays von CANopen-SDOs an.>;
    MAX_PDOINDEX_RX : INT :=            <Gibt an, wie viele RX-PDOs über alle CANopen-Master
                                        vorhanden sind.>;
    MAX_PDOINDEX_TX : INT :=            <Gibt an, wie viele TX-PDOs über alle CANopen-Master
                                        vorhanden sind.>;
    MAX_MASTER_ODENTRY_IDX : INT := <Gibt an, wie viele Objektverzeichniseinträge über alle
                                        CANopen-Master vorhanden sind.>;
END_VAR
```

Die Konfigurationsdaten werden in folgenden Arrays gespeichert:

```
VAR_GLOBAL
        pCANopenMaster :    ARRAY[0..MAX_MASTERINDEX] OF CanOpenMaster;
        pCanOpenNode :      ARRAY[0..MAX_NODEINDEX] OF CanOpenNode;
        pCanOpenSDO :       ARRAY[0..MAX_SDOINDEX] OF CanOpenSDO;
        pCanOpenPDO_Rx :    ARRAY[0..MAX_PDOINDEX_RX] OF CanOpenPDO_Rx;
        pCanOpenPDO_Tx :    ARRAY[0..MAX_PDOINDEX_TX] OF CanOpenPDO_Tx;
        ODMEntries:         ARRAY[0..MAX_MASTER_ODENTRY_IDX] OF CanOpenODEntry;
END_VAR
```

Dabei sind die FB-Membervariablen wie folgt benutzt: (Variablen die rein interne Variablen sind, sich also in den VAR-Abschnitten befinden, sollten von der Applikation nicht beschrieben werden, einige davon sind jedoch zu Diagnosezwecken interessant.)

```
VAR_INPUT
    nStatus : INT;              (* Gibt den aktuellen Status des Masters an. Dieser Status hat nichts
                                mit den nach CANopen definierten Stati zu tun. Es handelt sich hier
                                um eigene interne Definitionen.*)
    bMsgUsed: BOOL;             (* Kann von der Applikation benutzt werden, um festzustellen wann
                                eine neue SDO-Client Anfrage bearbeitet wurde. Dann kann z.B. das
                                Objektverzeichnis des Masters auf Änderungen untersucht werden,
                                oder insgesamt neu ausgewertet werden. (Gilt nur in Verbindung mit
                                den entsprechenden Einträgen in der cfg-Datei von CoDeSys, so
                                dass der Master über ein Objektverzeichnis verfügt. Sonst ohne
                                Bedeutung.)*)
    nRxIndex : INT;             (* Historisch, um Übersetzungsfehler in alten Projekten zu vermeiden,
                                unbenutzt.*)
    wDrvNr : WORD;              (* Hier wird bei der Konfiguration die Nummer des CAN-Controllers
                                übergeben, die von der Bibliothek an den CAN-Treiber übergeben
                                wird, um dem Treiber mitzuteilen, auf welchen Controller sich die
                                Schreib- und Leseaufträge beziehen. Die Applikation darf diesen
                                Parameter nie beschreiben.*)
    bUseStartAllNodes : BOOL; (* Die Applikation kann hier festlegen, ob die Bibliothek den Befehl
                                ‚Start Node' oder ‚Start All Nodes' benutzen soll. Die Applikation darf
                                dieses Flag nur einmal beschreiben, möglichst während der ersten
                                Zyklen nach dem Hochlauf. Zyklisches Beschreiben führt dazu, dass
                                einzelne Slaves nach einem Ausfall nicht gestartet werden.*)
    bUseResetAllNodes : BOOL; (* Analog zu bUseStartAllNodes*)
    bAutoStart : BOOL;          (* Wird von CoDeSys eingetragen, wenn die Autostart-Option des
                                Masters gesetzt wurde. *)

END_VAR
VAR_OUTPUT
    bError : BOOL;              (* Historisch, wird nicht benutzt.*)
END_VAR
VAR (* Konfig *)
    sDrvName : STRING(40);  (* Historisch, wird nicht benutzt.*)
```

tech_doc_d.dot / V1.2

```
    wBaudrate : WORD;          (* Hier wird von CoDeSys die Baudrate aus der Konfiguration
                               eingetragen.*)
    nFirstNodeNr : INT;        (* Hier wird von CoDeSys der Index des ersten Slaves, der zu diesem
                               Master gehört, eingetragen.*)
    nLastNodeNr : INT;         (* Hier wird von CoDeSys der Index des letzten Slaves, der zu diesem
                               Master gehört, eingetragen.*)
    SyncTimer : TON;           (* Der Timer, der für die Erzeugung der Sync-Nachricht benutzt wird.*)
    dwCOBID_Sync : DWORD; (* Die COBID der Sync-Nachricht aus der Konfiguration wird hier von
                               CoDeSys eingetragen.*)
    dwHeartbeatTime : DWORD; (* Die Heartbeat-Erzeugungszeit in ms wird hier von CoDeSys
                               eingetragen.*)
    nNodeId : WORD;            (* Die NodeId des Masters aus der Konfiguration wird hier
                               eingetragen.*)
    HeartbeatTimer  :TON;      (* Dieser Timer wird vom Master zur Heartbeat-Erzeugung benutzt.*)
    bSendHeartbeat : BOOL;     (* Wird intern vom Master gesetzt und wieder gelöscht, immer wenn
                               eine Heartbeat-Nachricht verschickt werden soll und verschickt
                               wurde.*)
    byHeartbeatState : BYTE;   (* Hier werden je nach Status des Masters die CANopen-konformen
                               Stati 0, 4, 5 oder 127 eingetragen, die mit der Heartbeat-Nachricht
                               verschickt werden.*)
    wODMFirstIdx : WORD;       (* Der Index des ersten Master-OD-Eintrags im Array ODMEntries.*)
    wODMCount : WORD;          (* Anzahl der ODMEntries die zu diesem Master gehören.*)
END_VAR

VAR
    bReentry: BOOL;            (* Historisch, unbenutzt.*)
    bGefunden : BOOL;          (* Historisch, unbenutzt.*)
    dwCOBID_NMT : DWORD := 16#0000; (* Diese ID wird ausschließlich zum Starten der Slaves
                                       verwendet. Sollte von der Applikation nie geändert
                                       werden.*)
    nIndex : INT; (*  interne Hilfsvariable des Masters.*)
    dwMerker : DWORD;          (* Historisch, unbenutzt. *)
    dwSem : DWORD := 16#FFFFFFFF; (*   Vorgesehen für Verriegelungen im Multitask-Betrieb.
                                       Unbenutzt, da die impliziten Aufrufe immer nur aus
                                       einer Task erfolgen.*)
    a: INT; (*Lifecounter, wird nur hochgezählt, um im Monitorbetrieb die impliziten Aufrufe sichtbar
                                      zu machen. Hier kann geprüft werden, ob der Master
                                      aufgerufen wird.*)
    bSynchSend : BOOL;         (* Diese Variable wird immer, wenn der Master eine Sync-Nachricht
                               verschickt, für einen IEC-Zyklus zu TRUE. Dasselbe passiert auch,
                               wenn der Master einen extern erzeugten Sync empfängt. Wenn der
                               Master den Sync selber erzeugt, wird bei einem externen Sync das
                               Flag nicht gesetzt.*)
    bErrCodeNot0 : BOOL;       (* Wird benutzt, um den Errorcode des CAN-Treibers für BusOFF
                               (Errorcode = 1) mit einem Zyklus Versatz zu löschen.*)
    MsgBuffer: CAN_Message (* Empfangspuffer des Masters. Wird nur für die SDO-Server
                               Funktionalität und für den Empfang eines externen Sync benutzt.*);
    bSDOMsgUsed : BOOL         (* Wird immer dann gesetzt, wenn ein Zugriff auf das
                               Objektverzeichnis des Masters erfolgte.*);
    bSDOReadrqActive : BOOL; (* Die folgenden Variablen werden benutzt, um die Zugriffe auf das
                               Objektverzeichnis des Masters zu verwalten.*)
    bSDOWriterqActive : BOOL;
    bSub0NotFound, bSDOReadrspAbort, bInitiateRspSend, bSDOWriterspAbort,
                                bInitiateWrReqSend : BOOL;
    i, iActiveSegSDORead : INT;
    ucModus : BYTE;
    dwIdx : DWORD;
    wSegSDOReadSendOffs,wSegSDOReadSendLen : WORD;
    pActiveSegSDORead:POINTER TO BYTE;
    iActiveSegSDOWrite:INT := -1;
```

tech_doc_d.dot / V1.2

```
    pActiveSegSDOWrite:POINTER TO BYTE;
    wSegSDOWriteRecvOffs : WORD;
    wSegSDOWriteRecvLen : WORD;
    dwODEValue : DWORD;
    iSDOReadLen, iSDOWriteLen : INT;
    byAbortCode : ARRAY[0..3] OF BYTE;
    bSwap : BOOL;           (* Hier speichert der Master die Tatsache, ob er auf einer INTEL oder
                                Motorola-Byteorder-Maschine läuft.*)
END_VAR
```

Um also z.B. die Variable bUseStartAllNodes des ersten Masters zu setzen, muss man
pCanOpenMaster[0]. bUseStartAllNodes := TRUE;
schreiben.

2.4.2 Die Struktur des CANopen-Nodes (Slaves)

Der Zugriff auf die CANopen-Nodes erfolgt über die vom Programmiersystem implizit angelegten Variablen pCanOpenNode. Diese Variable ist ein Array von Funktionsblöcken, die die konfigurierten Nodes darstellen.

Methoden der Nodes, wie z.B. ResetNodes, werden also folgendermaßen aufgerufen:

```
pCanOpenNode[0].ResetNode();
```

Dabei ist „0“ der Index des Knotens und nicht die NodeID. Die NodeId ist Teil der Instanz des FBs.

Auf alle Instanzdaten eines Nodes kann lesend und (z.T.) schreibend (dazu öffnen Sie bitte die Bausteinschnittstelle im Bibliotheksverwalter von CoDeSys) zugegriffen werden.

Die Komponenten des FBs im Detail:

```
VAR_INPUT
    nRxIndex : INT;             (* historisch, unbenutzt.*)
    nSDOSend : INT;             (* Der Index des aktuellen SDOs, das in der Konfigurationsphase
                                   an den Node geschickt wird.*)
    bAutoStart : BOOL;          (* Diese Komponente wird immer gesetzt, wenn der Master den
                                   Node auf automatisch Starten setzt.*)
    nNewStatus: INT;            (* Diese Variable wird benutzt, um die Aktion SetNodeStatus mit
                                   diesem Parameter aufrufen zu können.*)
    bSynchSend: BOOL;           (* Wird vom Master beim Aufruf des Bausteins übergeben, um
                                   das  Eintreffen eines extern erzeugten Sync oder das Erzeugen
                                   eines Sync durch den Master anzuzeigen. Wird nur intern
                                   verwendet und sofort wieder gelöscht.*)
    dwHeartbeatTime : DWORD;    (* Das Intervall in ms, mit dem der Master den Erhalt einer
                                   Heartbeat-Message des Knotens überwacht. Vom Konfigurator
                                   wird diese Zeit mit dem 1,5fachen der Heartbeat-Zeit des Knotens
                                   eingetragen.*)
    bSendReset: BOOL := TRUE;   (* Wird vom Master immer gesetzt, außer der Master möchte
                                   “Reset All Nodes” benutzen, dann wird der Reset durch den
                                   Baustein abgeschaltet.*)
END_VAR
VAR (* Konfig *)
    wDrvNr : WORD;              (* Die Nummer des CAN-Controllers, mit dem dieser Baustein
                                   arbeitet.*)
    wMasterIdx : WORD;          (* Der Index des Masters, unter dem dieser Slave arbeitet.*)
    ucNodeNr : BYTE := 1;       (* Die NodeID des Slaves, aus der Konfiguration.*)
    dwNodeIdent : DWORD;        (* Hier wird der Typ des Nodes gespeichert, der im Objekt
                                   0x1000 des Nodes ebenfalls gespeichert ist.*)
    dwGuardCOBID : DWORD;       (* Die COBID des Guard-Telegramms, unter der dem Slave
                                   Nodeguard-Telegramme geschickt werden.*)
```

```
    dwEmergCOBID : DWORD;   (* Die Emergency-COBID, auf der Emergency Telegramme des
                             Slaves erwartet werden.*)
    wDiagSegment : WORD;    (* Dient in Zukunft zum Speichern der Diagnoseadresse.*)
    dwDiagOffset: DWORD;
    wFirstSDOIndex : WORD;  (* Der Bereich der Konfigurations-SDOs für diesen Slave
                             innerhalb des Arrays von SDOs wird hier vom Konfigurator
                             eingetragen .*)
    wNummOfSDO : WORD;      (* Anzahl der SDOs, die vom Konfigurator für diesen Slave
                             erzeugt wurden.*)
    wFirstRxPDOIndex : WORD; (* In diesem und den folgenden 3 Komponenten werden die PDOs
                             zu diesem Slave identifiziert.*)
    wNummOfRxPDO: WORD;
    wFirstTxPDOIndex: WORD;
    wNummOfTxPDO: WORD;
    GuardTime : TIME;       (* Die Nodeguard-Time aus der Konfiguration wird hier
                             eingetragen.*)
    GuardTimer : TON;       (* Der Timer wird zur Erzeugung der Guard-Nachricht benutzt.*)
    LifeTimer : TON;        (* Der Timer wird zur Überwachung des Slaves benutzt. Wenn
                             innerhalb des Lifetimers nicht wenigstens eine Antwort auf ein
                             Nodeguard-Telegramm kommt, wird ein Guard-Fehler
                             ausgegeben.*)
    bDoInit : BOOL := TRUE; (* Wird vom Konfigurator auf FALSE gesetzt, wenn die Option
                             “Nicht initialisieren eingestellt ist.*)
    bOptional :BOOL;        (* Wird vom Konfigurator auf TRUE gesetzt, wenn die Option
                             “Optionales Gerät eingestellt wurde.*)
    bHeartbeatConsumer: BOOL; (* unbenutzt. Für Informationszwecke reserviert.*)
    tHeartBeatLifeTime : TON; (* Überwachungszeit der Heartbeat-Messages eines Slaves. Wird
                             immer neu gestartet, wenn eine Heartbeat-Message vom Slave
                             empfangen wurde. *)
END_VAR
VAR_OUTPUT
    byLastState : BYTE := 0;  (* Hier wird der letzte Status, der in der Antwort des Nodes auf die
                             letzte Guard-Message enthalten war, gespeichert.*)
        nStatus : INT;          (* Hier wird der aktuelle Status des Nodes eingetragen. Die
                                 Stati sind Zustände des Nodes innerhalb dieser Bobliothek
                                 und haben nichts mit den Zuständen wie sie nach CanOpen
                                 definiert sind, zu tun.
                                 0: undefiniert.
                                 1: Der Node wird vom Master zurückgesetzt und neu
                                 konfiguriert. Der Master wartet auf eine Bootup-Nachricht des
                                 Nodes (oder auf den Ablauf der eingestellten GuardTime) und
                                 schaltet dann den Status des Nodes auf Status 2.
                                 2: Der Master wartet vor Anfrage des Objekts 0x1000 ca.
                                 300ms. Dann wird der Status auf 3 gesetzt.
                                 3: Der Master beginnt mit einem SDO-Read-Request (Index
                                 0x1000) an den Node die Konfiguration des Slaves. Alle vom
                                 Konfigurator erzeugten SDOs werden dem Node in diesem
                                 Status geschickt. Die erzeugten SDOs stehen alle in einem
                                 Array von SDOs, in dem der Node sein erstes SDO und die
                                 Anzahl seiner SDOs kennt. (Instanzkomponente
                                 „wFirstSDOIndex” und „wNummOfSDO“)
                                 4: Ist bei der Konfiguration des Masters Autostart aktiviert, so
                                 bekommt der Node ein "Start Node" geschickt und wechselt in
                                 Status 5..
                                 5: Der Node empfängt/sendet PDOs, der Master ebenfalls.
                                 Normalbetrieb.
                                 97: Optionaler Node wurde beim Hochlauf nicht detektiert.

                                 98: Der Node hat auf Anfrage des Gerätetyps (Objekt 0x1000)
                                 mit einem anderen Typ geantwortet.
```

tech_doc_d.dot / V1.2

```
                                    99: Der LifeTimer ist abgelaufen, Nodeguarding-Fehler.*)
                                    Der nomale Status des Nodes im Betrieb, in dem IODaten
                                    ausgetauscht werden, ist 5.
    nSDOActiv : INT;            (* Hier verwaltet der Knoten den Index des aktiven SDO's (-1
                                bedeutet kein SDO-aktiv) *)
    EmcyMsg : CAN_Message;      (* Hier wird die letzte Emergency-Nachricht, die vom Node
                                empfangen wurde, gespeichert.*)
    sdoConfig : CanOpenSendSDO; (*Instanz des Funktionsblocks aus der CanManger-Lib, die zum
                                Konfigurieren des Nodes benutzt wird.*)
END_VAR
VAR
    nSDOIndex : INT;            (* Aktueller Index der Konfigurations-SDOs.*)
    bReady : BOOL; (*           Immer wenn eine Emergency oder Guard Nachricht empfangen
                                wurde wird bReady auf TRUE gesetzt, einen Zyklus lang.*)
    SDO_TimeOut: TON := ( PT := T#500ms );
    dwMerker : DWORD;           (* Hier werden die unteren 16 Bit des Objekts 0x1000, wie sie vom
                                Gerät gelesen wurden gespeichert.*)
    nStatusOld : INT;           (* Interne Variable um einen Statuswechsel zu erkennen.*)
    bManualStart : BOOL := FALSE; (*Wird intern von StartNode benutzt, um dem Funktionsblock zu
                                signalisieren, dass der Slave jetzt gestartet werden soll.*)
    bDevTypeInvalid : BOOL;     (* Der Gerätetyp, mit dem der Node auf Anfrage des Objekts
                                0x1000 geantwortet hat, ist nicht mit dem in der
                                Steuerungskonfiguration identisch. Durch manuelles (oder durch
                                das IEC-Program) Setzen des Node-Status (der, wenn
                                bDevTypeInvalid gleich TRUE ist, auf 98 stehen sollte) auf 4 kann
                                der Node trotzdem in Betrieb genommen werden. Die Applikation
                                kann nicht auf die Variable nStatus eines Nodes schreiben. Dazu
                                muss die Methode „NodeStart“ oder „SetNodeStatus(<new state>)“
                                der Nodes benutzt werden.*)
    MsgBuffer : CAN_Message;    (* Der Empfangspuffer des Nodes. Intern, nur für spezielle
                                Nachrichten benutzt.*)
    a: INT;                     (* Der Lebenszähler um anzuzeigen ob der Baustein aufgerufen
                                wird.*)
    iPDO : WORD;                (* Interne Variable, wird benutzt um alle zu diesem Slave
                                gehörenden PDOs zu durchsuchen.*)
END_VAR
```

2.4.3 Zustände der Bibliothek interpretieren und reagieren

Beim Hochlauf des CANopen Netzwerks und während des Betriebs durchläuft die Bibliothek, oder besser gesagt, die einzelnen Funktionsblöcke, verschiedene Zustände.

Im Monitorbetrieb von CoDeSys können diese Zustände in der globalen Variablenliste „Can Open implicit variables“ eingesehen werden. Hier sind jeweils in Arrays von Funktionsblöcken die im System konfigurierten Master und Nodes untergebracht.

Hier werden die Zustände der Variablen beschrieben, um einige Standardsituationen zu erkennen. Die Reaktionen darauf sind Sache des Anwenderprogramms.

2.4.3.1 Hochlauf des CANopen-Masters

Während des Hochlaufs des CAN-Netzwerks durchläuft der Master verschiedene Zustände, die vor allem in der Variablen nStatus abzulesen sind.

Die Stati **0**, **1** und **2** werden vom Master automatisch und in den ersten Zyklen nach einem SPS-Start durchlaufen. Der Status **3** des Masters wird für einige Zeit beibehalten. Im Status 3 konfiguriert der Master seine Slaves. Dazu werden den Slaves der Reihe nach alle vom Konfigurator erzeugten SDOs gesendet. Nachdem den Slaves alle Konfigurations-SDOs übertragen wurden, geht der Master nach Status **5** und bleibt in diesem Status. Status 5 ist für den Master der normale Betriebszustand. Der Master bleibt, nachdem er diesen Status einmal erreicht hat, im Status 5. Alle anderen Zustände werden von den Funktionsblöcken, die die Slaves darstellen, verwaltet.

tech_doc_d.dot / V1.2

Immer, wenn ein Slave auf eine SDO-Anfrage (Upload oder Download) nicht antwortet, so wird sie vom entsprechenden Funktionsblock wiederholt. Der Master verlässt den Status 3, wie bereits erwähnt, aber erst, wenn alle SDOs erfolgreich übertragen wurden. So kann also erkannt werden, ob ein Slave fehlt oder nicht alle SDOs richtig empfangen kann. (Dabei ist es für den Master unerheblich, ob ein Slave mit einer Bestätigung oder einem Abort antwortet, für den Master ist nur die Tatsache von Interesse, ob überhaupt eine Antwort empfangen wurde. Ein Beispiel zur Erkennung von Aborts während des Hochlaufs siehe unten, Kapitel 2.4.3.2.)

Eine Ausnahme stellt ein als optional markierter Slave dar. Optionale Slaves werden nur einmal nach ihrem 0x1000er-Objekt gefragt. Wenn sie nicht innerhalb von 0,5 sec. antworten wird der Slave vom Master zunächst ignoriert und der Master geht auch ohne weitere Reaktion dieses Slaves in Status 5. Soll dieser Slave von da an nochmals vom Master konfiguriert oder auf Anwesenheit getestet werden, muss die Applikation das veranlassen. (Siehe Zusatz in Kapitel 2.4.4.2, Hochlauf des Netzwerks ohne automatisches Starten.)

2.4.3.2 Hochlauf der CANopen-Slaves

Ein Slave wird repräsentiert durch eine Funktionsblockinstanz im Array pCanOpenNodes in der globalen Variablenliste „Can open implicit Variables“. Während des Hochlaufs des Can-Netzwerks durchläuft der Slave die Stati -1, 1 und 2 automatisch. Dabei sind diese Stati wie folgt zu interpretieren: (Es handelt sich hier zum Teil um interne Stati, die von der Applikation u.U. nicht alle gesehen werden, da sie nur einen Zyklus lang anstehen.)

- **-1** Node wird durch die NMT-Nachricht „Reset Node“ zurückgesetzt und wechselt selbständig in den Status 1.
- **1** Node wechselt nach einer maximalen Zeit von 2 sec. oder sofort nach Empfang seiner Bootup-Message in den Status 2.
- **2** Der Node wechselt nach einer Verzögerungszeit von 0,5 sec. automatisch in den Status **3**. Diese Zeit entspricht der Erfahrungstatsache, dass viele CANopen-Geräte nicht sofort nachdem sie Ihre Bootup-Message verschickt haben, bereit sind, ihre Konfigurations-SDOs zu empfangen. Im Status 3 wird der Slave konfiguriert.

Der Slave bleibt solange im Status 3, bis er alle vom Konfigurator erzeugten SDOs erhalten hat. Dabei spielt es keine Rolle, ob während der Konfiguration vom Slave SDO-Transfers mit Abort (Fehler) oder ob alle fehlerfrei beantwortet wurden. Nur die Tatsache selbst, dass eine Antwort vom Slave erhalten wurde, spielt eine Rolle. Die Applikation kann die Antworten auf die SDO-Anfragen überwachen und selbst auf die Antworten der Slaves reagieren.

Wenn die Option „Knoten zurücksetzen“ im Konfigurator aktiviert wurde, wird nach dem Senden des Objekts 0x1011 Subindex 1, der dann den Wert „load“ enthält, ein erneuter Reset des Nodes durchgeführt. Dabei trägt die Bibliothek in dieses Objekt 0 ein, damit der Reset nur einmal nach der Initialisierung der Anwendung ausgeführt wird. Der Slave wird dann wieder mit dem Upload des Objekts 0x1000 angefragt.

Nachdem der Slave die Konfigurationsphase durchlaufen hat, kann er in folgende Stati übergehen:

- Er wechselt immer in Status **4**, außer es handelt sich um einen optionalen Slave und er wurde nicht als am Bus verfügbar detektiert (Abfrage Objekt 0x1000), oder der Slave ist zwar vorhanden, aber hat auf die Abfrage des Objekts 0x1000 mit einem anderen Typ in den unteren 16 Bit reagiert, als der Konfigurator erwartet hat.
- Er wechselt in den Status **97** wenn er optional ist (Option „Optionales Gerät“ in der Can Konfiguration) und nicht auf die SDO-Anfrage nach dem Objekt 0x1000 reagiert hat.
- Er wechselt nach Status **98** wenn der Gerätetyp (Objekt 0x1000) nicht dem konfiguriertem entspricht, nachdem er alle Konfigurations-SDOs trotzdem bekommen hat. (Zum Starten des Nodes in diesem Fall siehe Zusatz in 2.4.4.2 ’Hochlauf des Netzwerks ohne automatisches Starten’.)

Wenn der Master auf automatisches Starten konfiguriert wurde, wird der Slave im Status 4 gestartet (d.h. es wird eine „Start Node“-NMT-Nachricht erzeugt) und der Slave wechselt automatisch nach Status **5**. Status 5 ist der normale Betriebszustand des Slaves. Wurde von der Applikation das Masterflag bUseStartAllNodes gesetzt, dann wird gewartet, bis sich alle Slaves im Zustand 4 befinden und dann werden alle Slaves mit dem NMT-Kommando „Start All Nodes“ gestartet.

tech_doc_d.dot / V1.2

Befindet sich der Slave im Status 4 oder höher, werden Nodeguard-Nachrichten an den Slave gesendet, wenn Nodeguarding konfiguriert wurde.

2.4.3.3 Nodeguarding-/Heartbeatfehler

Im Falle eines Nodeguarding-Timeouts wird die Variable nStatus des Nodes auf 99 gesetzt. Um den Slave wieder zu starten, genügt es, den Status des Nodes wieder auf 4 zu setzen. Damit wird dem Node ein „Start communication“ geschickt, der CANopen-Master beginnt wieder zu kommunizieren. Dazu wird die Methode „NodeStart“ der Nodes benutzt.

Der Master macht dieses von sich aus, sobald der Node wieder auf Nodeguard-Anfragen reagiert und die Option „Autostart“ eingeschaltet ist. Dabei wird der Node abhängig von seinem Status, der ja in der Antwort auf die Nodeguard-Anfragen enthalten ist, neu konfiguriert, oder nur gestartet.

Ohne die Option AutoStart muss die Applikation die Methode NodeStart aufrufen. Je nach letztem gespeicherten Status (der ja immer mit der Heartbeat-/Nodeguard-Message übertragen wird), wird der Knoten neu gestartet oder zurückgesetzt.

Für Heartbeat-Fehler gilt dasselbe Vorgehen.

2.4.3.4 Node antwortet nicht in Konfigurationsphase

Die Applikation kann die Konfigurationsphase eines Slaves zeitüberwachen, und Nodes, die im Status 3 bleiben auf „TimeOut“ setzen. Die Applikation kann dies durch Aufrufen der Methode „NodeSetTimeoutState“ des Nodes erreichen (also z.B. pCanOpenNode[0]. NodeSetTimeoutState();).

Mit NodeSetTimeoutState wird der Status des Nodes auf 97 gesetzt, damit kann anschließend so verfahren werden, als ob es sich um einen optionalen Knoten handelt.

Wenn alle konfigurierten Nodes einen Status > 4 erreicht haben, geht der Master in den Status 5, Normalbetrieb, über. In diesem erfolgt ein Austausch von Prozessdaten über PDOs.

2.4.3.5 Antworten der Slaves auf die Konfigurations-SDOs auswerten

Die Applikation kann die gesamte Konfigurationsphase eines Nodes mitprotokollieren, also jede Antwort „sehen“. Die Antworten auf die Konfigurations-SDOs stehen in sdoConfig der Node-Struktur: zum Beispiel in

pCanOpenNode[0].sdoConfig.ucAnswerBytes[0..7]
(siehe dazu auch die Beschreibung des Funktionsblocks CanOpenSendSDO in Kapitel 2.4.4.1.)

Der Master sendet ein SDO nach dem anderen, sobald eine Antwort vom Slave erhalten wurde. Deshalb muss die Applikation die Änderung der Antwort erkennen, um den Zeitpunkt zu bestimmen, wann eine neue SDO-Antwort eingetroffen ist:

```
VAR
        ucOldAnswer : array[0..7] of BYTE;
        i : INT;
VAR_VAR
FOR i := 0 to 7 DO
    IF pCanOpenNode[0].sdoConfig.ucAnswerBytes[i] <> ucOldAnswer[i] THEN
        ucOldAnswer := pCanOpenNode[0].sdoConfig.ucAnswerBytes;
        (*Hier kann die neue Antwort ausgewertet werden, wenn die Applikation das will.*)
        (*Zu den Abortcodes siehe Beschreibung des CanOpenSendSDO
          in Kapitel 2.4.4.1.*)
        EXIT; (*Schleife bei der ersten Änderung verlassen.*)
    END_IF
END_FOR
```

2.4.4 Beispiele für Applikationsgesteuerte Aktionen der Bibliotheken

Hier sollen einige standardmäßige Aktionen erläutert werden, die Applikationen oft durchführen müssen.

tech_doc_d.dot / V1.2

2.4.4.1 SDO-Transfer über den Funktionsblock „CanOpenSendSDO“

Um einen SDO-Transfer applikationsgesteuert zu initiieren, benutzen Sie den Baustein CanOpenSendSDO der CanOpenManager-Bibliothek.

Die Schnittstelle des Bausteins:

```
VAR_INPUT
        Enable  : BOOL;         (*Eine steigende Flanke an diesem Flag startet die SDO-
                                Übertragung.*)
        wDrvNr : WORD;          (*Index des Can Controllers der für den SDO-Transfer benutzt
                                werden soll. 0 für den ersten und so weiter.*)
        ucNodeId : BYTE;        (*NodeId des SDO Servers. (Empfänger des SDO)*)
        wIndex  : WORD;         (*Index des Objektverzeichniseintrags.*)
        bySubIndex : BYTE;      (*SubIndex*)
        ucModus: BYTE;          (* SDO-mode, 16#40 für read-request,
                                        use 16#23 für 4-byte-write-request.
                                        use 16#27 für 3-byte...
                                        use 16#2B für 2-byte...
                                        use 16#2F für 1-byte...
                                        use 16#21 für einen Download mit mehr als 4 Bytes.*)
        ucByte0  : BYTE;        (*Die 4 möglichen Datenbytes beim „expedited” Transfer. *)
        ucByte1  : BYTE;
        ucByte2  : BYTE;
        ucByte3  : BYTE;

        aAbortCode : ARRAY[0..3] OF BYTE;

        (*Zusätzliche Eingabeparameter wenn der segmentierte Transfer benutzt werden soll.*)
        (*Für die Modi 16#41 und 16#40 für das segmentierte Lesen, und 16#21 für segmentiertes
        Schreiben. In diesen Fällen muss ein Puffer für die Daten angegeben werden.*)
        dwDataBufferLength : DWORD;
        pDataBuffer : POINTER TO BYTE;
END_VAR

VAR_OUTPUT
        bWaitForAnswer : BOOL; (* Dieses Flag wird während der Übertragung mindestens einen
                                  Zyklus lang zu TRUE gesetzt und nach der Übertragung zu
                                  FALSE.*)
        bAnswerRec : BOOL;   (*Nur bei einer fehlerfreien Übertragung wird dieses Flag zu TRUE.*)
        ucAnswerBytes : ARRAY[0..7] OF BYTE;
        iAnswerLength : INT;
        bToggleUnequal: BOOL;
        bAbortRec:BOOL;         (* Wenn ein Abort stattgefunden hat, wird diese Variable zu TRUE,
                                der Abortcode steht in aAbortRec.*)
        aAbortRec : ARRAY[0..3] OF BYTE;
END_VAR

VAR
        EnableOld : BOOL;
        bAnswer : BOOL;
        n: INT;
        dwActiveCOBId : DWORD;
        dwAnswerId : DWORD;
        Buffer : CAN_Message;
        bExpedited : BOOL;              (* Hier zeigt der Baustein an, ob beim Lesen ein „expedited“
                                        oder segmentierter Transfer benutzt wurde.*)
        bWriteActive : BOOL;
        bReadActive : BOOL;
        dwDataOffset : DWORD;           (* Nach dem Transfer steht die tatsächlich  gelesenen bzw.
                                        geschriebene Anzahl von Bytes in dieser Variablen.*)
```

tech_doc_d.dot / V1.2

```
        dwDataReadLength : DWORD; (* Hier wird die voraussichtliche Länge eines SDO-Uploads
                                     (aus der 1. Antwortnachricht) abgelegt.*)
        ucUploadRequest: BYTE;
        bLastToggle: BOOL;
        iCurSDODataLen : INT;
        ucDownloadRequest: BYTE;
END_VAR
```

Um den Baustein zu benutzen, muss für jede gleichzeitige Übertragung eine statische Instanz des Bausteins angelegt werden (statisch heißt: nicht als lokale Variable einer Funktion). Es ist nicht möglich (per CANopen-Definition) gleichzeitig mehrere SDO-Kanäle zum gleichen Slave hin zu öffnen.

SDO senden (Expedited Mode):

Als Parameter werden dem Baustein die Adresse (NodeID) des Ziels, die (0-basierte) Nummer des CAN-Netzes, auf dem der Baustein arbeiten soll, die 4 Datenbytes, Index und Subindex des Zielobjekts übergeben. Außerdem benötigt der Baustein den Modus, in dem er arbeiten soll: Zum Senden von

- 4 Bytes muss Mode 16#23,
- 3 Bytes muss Mode 16#27,
- 2 Bytes muss Mode 16#2B,
- 1 Byte muss Mode 16#2F

übergeben werden.

Der Aufruf des Bausteins, um einen 4-Byte-Wert in einen SDO-Server zu schreiben könnte also z.B. so aussehen:

```
        sdo(
        Enable:= TRUE,
        wDrvNr:= 0,
        ucNodeId:= 2,
        wIndex:= 16#4002,
        bySubIndex:= 0,
        ucModus:= 16#23,
        aAbortCode:=aAbort,
        ucByte0 := DWORD_TO_BYTE(dwWrite),
        ucByte1 := DWORD_TO_BYTE(SHR(dwWrite,8)),
        ucByte2 := DWORD_TO_BYTE(SHR(dwWrite,16)),
        ucByte3 := DWORD_TO_BYTE(SHR(dwWrite,24)));
```

Dabei ist sdo eine Instanz von CanOpenSendSDO.

Die Überprüfung, ob der Transfer beendet wurde, sieht z.B. so aus:

```
        IF (sdo.bAnswerRec OR NOT sdo.bWaitForAnswer) THEN
                ….. (*Der Transfer ist beendet. Wenn bAnswerRec nicht TRUE ist, trat ein Fehler
                auf.*)
        END_IF
```

SDO empfangen:

Wie SDO senden, nur dass als Datenbytes 4 mal 0 übergeben wird und als Modus 16#40.

Jeder andere Übergabewert für „Mode“ führt zu einem undefinierten Verhalten des Bausteins.

Der Aufruf des Bausteins könnte also so aussehen:

```
        sdo(
```

tech_doc_d.dot / V1.2

```
Enable:= TRUE,
wDrvNr:= 0,
ucNodeId:= 2,
wIndex:= 16#4001,
bySubIndex:= 0,
ucModus:= 16#40,
aAbortCode:=aAbort);
IF sdo.bAnswerRec AND NOT sdo.bAbortRec THEN
        dwRead := SHL(BYTE_TO_DWORD(sdo.ucAnswerBytes[7]),24);
        dwRead := dwRead + SHL(BYTE_TO_DWORD(sdo.ucAnswerBytes[6]),16);
        dwRead := dwRead + SHL(BYTE_TO_DWORD(sdo.ucAnswerBytes[5]),8);
        dwRead := dwRead + BYTE_TO_DWORD(sdo.ucAnswerBytes[4]);
END_IF
```

Dabei wird davon ausgegangen, dass im Index 4001, Sub0, ein 4-Byte-Wert steht, der immer mit einem „Expedited“ gelesen werden kann.

Wenn man nicht weiß, ob der Wert vom SDO-Server mit einem „expedited“ Transfer übertragen wird oder nicht, muss immer ein Datenpuffer angegeben werden. Nach dem Transfer kann dann über das Flag bExpedited entschieden werden, ob die Daten in ucAnswerBytes[4..7] oder im Puffer stehen.

Der Eingabeparameter aAbortCode ist dazu da, dem Baustein den Abortcode, der gesendet werden soll, wenn die Kommunikation von der Applikation durch Löschen des Enable-Eingangs abgebrochen wird, mitzuteilen. (Immer wenn während einer laufenden Übertragung eine fallende Flanke am Enable Eingang vom Baustein gesehen wird, wird ein Abort erzeugt.)

Der Baustein macht von sich aus keine Timeout-Überwachung. Die Applikation muss also den SDO Transfer überwachen und z.B. mit dem Aufruf

Sdo(Enable := FALSE, aAbortCode[0]:=0,aAbortCode[1] := 0,aAbortCode[2] := 4,aAbortCode[3] := 5);

den Abortcode „SDO Protocol timed out“ übergeben, der mit einer fallenden Flanke an Enable verschickt wird, wenn das Protokoll noch nicht mit der Übertragung fertig war.

Sollen mehr als 4 Byte gelesen werden, kann dem Baustein auch eine Pufferadresse und ein maximale Größe des Puffers mitgegeben werden:

```
sdo(
Enable:= bReadString,
wDrvNr:= 0,
ucNodeId:= 2,
wIndex:= 16#4000,
bySubIndex:= 2,
ucModus:= 16#40,
aAbortCode:=aAbort,
dwDataBufferLength:= SIZEOF(str) ,
pDataBuffer:= ADR(str));
```

Dabei ist im Beispiel str ein String, dessen Größe im Speicher hier mit dem Sizeof-Operator ermittelt wird. So viele Zeichen werden vom Baustein maximal übertragen. Die tatsächlich gelesene Anzahl von Bytes steht nach dem Transfer in dwDataOffset zur Verfügung.

Allgemeines:

Der Baustein sendet die Anfrage immer auf der COBID: 16#600 + NodeId, die Antwort des SDO-Servers wird immer auf der COBID 16#580 + NodeId erwartet.

Die Antwortdaten sind transparent, d.h. es steht einfach die Antwortnachricht auf die SDO-Anfrage in ucAnswerBytes. Das gilt nur für die Antwort im Falle eines expedited Read Requests (Expedited Upload). Die Bedeutung der einzelnen Bytes ist in der Protokollspezifikation v. CANopen erläutert:

- Das Byte ucAnswerBytes[0] enthält den Server Command Specifier (ssc): 0x60 als Antwort auf einen Download Request, 0x4y als Antwort auf einen Upload Request (y: Bits 0..3 sind hier wie folgt belegt: Bit 1 muss 1 sein, ansonsten antwortet der SDO-Server hier mit einem segmentierten Upload, der vom Baustein nicht unterstützt wird). 0x80 bedeutet, dass der Transfer abgebrochen wurde (Abort).

tech_doc_d.dot / V1.2

- Die Bytes ucAnswerBytes[1..2] enthalten den Index. (Intel-byte-order)
- Das Byte ucAnswerBytes[3] enthält den SubIndex.

Die Bytes ucAnswerBytes[4..7] enthalten die Daten der Antwort, im Falle eines „expedited“ SDO-Uploads.
Im Falle eines Abbruchs steht der Abortcode in aAbortRec, siehe nachstehende Tabelle (aus DSP3.01V401):
0503 0000h Toggle bit not alternated.
0504 0000h SDO protocol timed out.
0504 0001h Client/server command specifier not valid or unknown.
0504 0002h Invalid block size (block mode only).
0504 0003h Invalid sequence number (block mode only).
0504 0004h CRC error (block mode only).
0504 0005h Out of memory.
0601 0000h Unsupported access to an object.
0601 0001h Attempt to read a write only object.
0601 0002h Attempt to write a read only object.
0602 0000h Object does not exist in the object dictionary.
0604 0041h Object cannot be mapped to the PDO.
0604 0042h The number and length of the objects to be mapped would exceed PDO length.
0604 0043h General parameter incompatibility reason.
0604 0047h General internal incompatibility in the device.
0606 0000h Access failed due to an hardware error.
0607 0010h Data type does not match, length of service parameter does not match
0607 0012h Data type does not match, length of service parameter too high
0607 0013h Data type does not match, length of service parameter too low
0609 0011h Sub-index does not exist.
0609 0030h Value range of parameter exceeded (only for write access).
0609 0031h Value of parameter written too high.
0609 0032h Value of parameter written too low.
0609 0036h Maximum value is less than minimum value.
0800 0000h general error
0800 0020h Data cannot be transferred or stored to the application.
0800 0021h Data cannot be transferred or stored to the application because of local control.
0800 0022h Data cannot be transferred or stored to the application because of the present device state.
0800 0023h Object dictionary dynamic generation fails or no object dictionary is present (e.g. object dictionary is generated from file and generation fails because of an file error).

2.4.4.2 Hochlauf des Netzwerks ohne automatisches Starten

Manchmal ist es notwendig, dass die Applikation den Zeitpunkt bestimmt, wann die CANopen-Slaves gestartet werden. Dazu muss die Option „Automatisch starten“ des CAN-Masters in der Konfiguration deaktiviert werden. Die Applikation ist dann für das Starten der Slaves zuständig.

Der normale Weg, einen Slave über die Bibliothek zu starten, ist, die Methode NodeStart des jeweiligen Slaves aufzurufen. Der Aufruf dieser Methode nach der Konfigurationsphase führt bei Slaves, für die Nodeguarding bzw. Heartbeat aktiv ist zu dem Ergebnis, dass an diese Slaves abhängig von ihrem aktuellen Status, der in der Nodeguard/Heartbeat-Nachricht von dem Slave enthalten ist, gestartet werden. Wenn der aktuelle Status des Slaves nicht bereits „OPERATIONAL“ ist, wird ein „Start Node“-NMT-Kommando an den Slave geschickt.

Wenn die Methode benutzt wird um einen Slave, der kein Nodeguarding benutzt, zu starten, wird dem Slave immer der komplette SDO-Satz neu geschickt und anschließend ein „StartNode“-Kommando abgesetzt, wenn der Hochlauf in Ordnung war (d.h. wenn der Gerätetyp mit der Konfiguration übereinstimmt). Um solche Slaves nach der automatischen Konfigurationsphase zu starten ist es einfacher selbst ein NMT-Kommando zu erzeugen. Hierdurch kann verhindert werden, dass den Slaves die Konfigurations-SDOs noch einmal geschickt werden. Zur Erzeugung des NMT-Kommandos kann ein Aufruf der CanOpenWriteMSG-Funktionsblockinstanz benutzt werden (Diese Instanz ist immer global in der Manager-Bibliothek vorhanden.)

tech_doc_d.dot / V1.2

```
CanOpenWriteMSG(  wDrvNr := pCanOpenNode[xx].wDrvNr, dwCanID := 0,
                  ucLen := 2, bRtrFrame := FALSE,
                  ucByte1 := 16#01, ucByte2 := pCanOpenNode[xx].ucNodeNr);
```

Damit wird immer ein StartNode-NMT-Kommando auf den Bus gelegt. Dabei werden die Parameter des Bausteins hier aus dem Array mit den Beschreibungen der Slaves entnommen.

Um das Netzwerk gesamt zu starten kann die NMT-Nachricht ebenfalls wie oben selbst generiert werden: (Diese Vorgehensweise muss immer für Nodes, die ohne Nodeguarding/Heartbeat arbeiten, benutzt werden.)

```
CanOpenWriteMSG(  wDrvNr := pCanOpenMaster[xx].wDrvNr, dwCanID := 0,
                  ucLen := 2, bRtrFrame := FALSE,
                  ucByte1 := 16#01, ucByte2 := 0);
```

Dieses NMT-Kommando heißt „Start All Nodes“.

Dabei müssen allerdings alle Stati der Nodes angepasst werden, damit die Bibliothek auch ohne dass der Master die Nodes startet, auf den aktuellen Status setzt.

Besser ist es hier die Methode StartAllNodes des Masters zu benutzen. Dabei wird das NMT-Kommando Start All Nodes erzeugt und die Stati aller Nodes werden aktualisiert.

Um einen Slave zu starten, der während des Hochlaufs in den Status 98 (Gerätetyp stimmt nicht mit dem konfiguriertn überein) gesetzt wurde, genügt es die Methode SetNodeStatus(nNewStatus := 4) zu benutzen. Wenn der Master auf automatisch starten konfiguriert wurde, wird der Node jetzt gestartet. Sonst muss die Applikation jetzt noch die Methode NodeStart des Nodes aufrufen:

```
pCanOpenNode[iNodeXX].NodeStart();
```

Wenn der Gerätetyp eines Slaves, der ohne Nodeguarding oder Heartbeat arbeitet, nicht mit der Konfiguration übereinstimmt, muss dieser durch explizites Senden einer NMT Message gestartet werden (siehe oben.) Hier führt der Aufruf der Methode NodeStart immer wieder zu einem Senden aller Konfigurationsdaten, ohne dass der Node gestartet wird.

Für einen optionalen Slave, der während des Hochlaufs des Masters nicht vorhanden war, also jetzt im Status 97 steht, kann man genauso vorgehen. Man kann aber auch einfach die Methode NodeReset aufrufen, die immer dazu führt, dass der Slave neu konfiguriert und dann gestartet wird. (Gestartet wird der Node nur, wenn die AutoStart-Option aktiviert ist. Sonst muss der Node auch noch manuell gestartet werden.)

2.4.4.3 Stoppen eines Nodes

Mit dem Aufruf der Methode NodeStop, analog zu NodeStart, wird ein NMT-Kommando zum stoppen des Slaves erzeugt. Der Slave geht daraufhin normalerweise in den Zustand „Stopped“ über.

2.4.4.4 Zurücksetzen eines Nodes

Mit dem Aufruf der Methode ResetNode wird der betreffende Slave zurückgesetzt und neu konfiguriert.

2.4.4.5 Starten des Netzwerks mit bUseStartAllNodes

In einem CAN-Netz mit vielen Teilnehmern (meist mehr als 8) kommt es häufig dazu, dass schnell aufeinander folgende NMT-Nachrichten nicht von allen (meist langsamen) IO-Knoten (Slaves) erkannt werden. Das liegt daran, dass diese Knoten alle Nachrichten mit der ID 0 mithören müssen. In zu schneller Folge gesendete NMT-Nachrichten überlasten die Empfangslogik solcher Knoten. Ein Symptom für diese Tatsache ist, dass diese Knoten dann manchmal nicht gestartet werden. Abhilfe kann man nur schaffen, wenn die Anzahl schnell aufeinander folgender NMT-Nachrichten reduziert wird. Dazu kann die Applikation die CANopen-Master-Bibliothek veranlassen, dass diese den Befehl Start All Nodes benutzt, anstatt alle Nodes einzeln mit dem Kommando „Start Node“ zu starten.

Dazu muss die Applikation einmalig, immer beim Anlauf der Steuerung, das Flag bUseStartAllNodes des Masters setzen, z.B. so (bInit ist von der Applikation definiert):

```
IF NOT bInit THEN
        pCanOpenMaster[0].bUseStartAllNodes := TRUE;
```

tech_doc_d.dot / V1.2

```
        bInit := TRUE;
    END_IF
```

Achtung: Dieses Flag darf nicht zyklisch gesetzt werden, da sonst einzelne Knoten nicht mehr richtig nach einem Ausfall gestartet werden.

Immer, wenn die Applikation diesen Befehl zum Starten des Netzwerks benutzt, werden auch Nodes, die z.B. weil sie sich mit dem falschen Gerätetyp gemeldet haben, in den Status 98 gesetzt wurden, gestartet. Die PDOs für diese Nodes bleiben jedoch weiterhin deaktiviert. Um diese freizuschalten, muss die Applikation wie in Kapitel 2.4.4.2, ‚Hochlauf des Netzwerks ohne automatisch Starten' beschrieben vorgehen. Wenn man sich nicht sicher ist, ob die Slaves alle mit dem richtigen Gerätetyp parametriert wurden, kann der Member „dwMerker" der einzelnen Nodes betrachtet werden. Hier wird der Gerätetyp, der vom Slave über die Abfrage des Objekts 0x1000 gelesen wurde, gespeichert. (Im Member dwNodeIdent wird die konfigurierte gespeichert.) dwMerker ist immer gültig, wenn er ungleich 0 ist.

2.4.4.6 Initialisieren des Netzwerks mit bUseResetAllNodes

Aus den selben Gründen wie bUseStartAllNodes gibt es Fälle in denen man besser das NMT-Kommando Reset All Nodes als Reset Node für jeden Node einzeln benuzt.

Dazu muss die Applikation einmalig das Flag bUseResetAllNodes des Masters setzen:

```
    IF NOT bInit THEN
        pCanOpenMaster[0]. bUseResetAllNodes := TRUE;
        bInit := TRUE;
    END_IF
```

Achtung: Dieses Flag darf nicht zyklisch gesetzt werden, da sonst einzelne Knoten nicht mehr richtig nach einem Ausfall zurückgesetzt werden.

2.4.4.7 Überwachen der Konfigurationsphase

Die Konfigurationsphase der Slaves kann überwacht werden. Zunächst einmal muss der Master in seiner Hochlaufphase überwacht werden, da er bei nStatus = 3 „hängen bleibt", wenn ein nicht-optionaler Slave während des Hochlaufs nicht vorhanden ist.

Slaves, bei denen während der Konfigurationsphase ein Problem auftritt, bleiben mit nStatus = 3 hängen, oder wechseln nach der Konfigurationsphase direkt in einen Fehlerstatus nStatus > 5.

Die folgenden Beispiele sind Applikationscode, um in bestimmten Situationen trotzdem mit dem Can-Netz arbeiten zu können, bzw. Diagnoseinformationen zu erhalten.

Bei der Überwachung der Hochlaufphase ist zu beachten, dass pro IEC-Zyklus der Task, in der der CANopenMaster aufgerufen wird (siehe auch Kapitel 2.5, Implizite Aufrufe) nur ein SDO an jeden Slave übertragen werden kann. D. h. entsprechend lange kann die Hochlaufphase dauern, je nach maximaler Anzahl der an einen Slave zu übertragenden SDOs.

2.4.4.7.1 Zugriff auf den Status des CANopen-Masters

Um zu verhindern, dass Applikationscode abgearbeitet wird, solange das IO-Netzwerk noch nicht bereit ist, muss der Status des Masters abgefragt werden:

```
    IF pCanOpenMaster[0].nStatus = 5 THEN
        <Applikationscode>
    END_IF
```

2.4.4.7.2 Zeitüberwachung der Hochlaufphase

```
    VAR
        Ueberwachung : TON;
        iSlave : INT;
        bSlaveFound: BOOL;
    END_VAR
    Ueberwachung(IN:= pCanOpenMaster[0].nStatus < 5, PT := T#10s );
    IF Ueberwachung.Q THEN
    (*ermitteln, ob es einen Slave gibt, der nicht konfiguriert werden kann.*)
```

tech_doc_d.dot / V1.2

```
            FOR iSlave := pCanOpenMaster[0]. nFirstNodeNr TO pCanOpenMaster[0].
            nLastNodeNr DO
                IF pCanOpenNode[iSlave]. nStatus < 4 THEN
                    bSlaveFound := TRUE;
                    EXIT;
                END_IF
            END_FOR
        END_IF
```

Nach dieser Sequenz steht in der Variablen iSlave die Nummer (Index) des ersten Slaves, der nach Ablauf der Überwachungszeit noch im Status 3 steht.

Dieses Beispiel gilt nur für einen nicht-optionalen Slave. Optionale Slaves müssen von der Applikation verwaltet werden, da nicht immer erkannt werden kann, ob diese Slaves vorhanden sind oder nicht. Die Hochlaufphase gilt auch beim Fehlen optionaler Slaves als o.k.

2.5 Implizite Aufrufe

CoDeSys erzeugt implizit Aufrufe an die CANopen-Bibliotheken.

Wenn CoDeSys mit der Kommandozeilenoption /debug gestartet wird, werden diese Aufrufe ins Übersetzungsverzeichnis als Dateien (Name beginnt mit CanOpen…., Endung ist .exp) gespeichert.

Die Aufrufe werden nach folgendem Schema generiert:

Gibt es keinen Modulparameter des CAN-Masters UpdateTask, dann werden alle Aufrufe in der höchstprioren Task gemacht. Der CanMaster und alle PDOs die zu diesem Master gehören, werden in einer Task aufgerufen. Mit dem Parameter Update Task des CanMasters wird die dort eingegebene Task für die Aufrufe benutzt.

Wenn PDOs oder der Masteraufruf nach obigem Schema in einer Ereignis-Task aufgerufen werden, wird eine Warnung ausgegeben.

Um die Aufrufe des CAN-Masters an eine Task zu binden und die PDOs in den Tasks, in denen die IOs dieser PDOs referenziert werden, zu erzeugen, kann man in der Sektion des CAN-Masters in der cfg-Datei des Zielsystems einen Eintrag PDOsToReferenceTasks=TRUE erstellen. Dieser Eintrag darf nur in Absprache mit dem Hersteller der Steuerung gemacht werden.

2.6 Das Objektverzeichnis des CANopen-Masters

In manchen Fällen ist es hilfreich, wenn der CAN-Master über ein eigenes Objektverzeichnis verfügt, z.B. zum Datenaustausch der Applikation mit anderen CAN-Knoten.

Das Objektverzeichnis des Masters wird über eine EDS-Datei zur Übersetzungszeit erstellt und mit Werten vorbelegt.

Die EDS-Datei wird über zwei Einträge in der cfg-Datei festgelegt:

MasterEDS=TRUE

MasterEDSFile=MyMaster.eds

Hier wird die erste, in allen Konfigurationsverzeichnissen gefundene Datei mit dem Namen MyMaster.eds benutzt.

Das Objektverzeichnis ist ein Array der folgenden Struktur:

```
TYPE CanOpenODEntry :
STRUCT
    dwIdxSubIdxF : DWORD;   (* Die Struktur dieser Komponente ist 16#iiiissff, wobei iiii für 2 Bytes
                               Index, ss für ein Byte Subindex und ff für 1 Byte Flags steht. *)
    dwContent : DWORD;      (* Der Inhalt des Eintrags.*)
    wLen : WORD;            (* Die Länge der Daten.*)
    byAttrib : BYTE;        (* Ursprünglich als Zugriffsberechtigung gedacht, kann von der
                               Applikation des Masters beliebig genutzt werden.*)
```

tech_doc_d.dot / V1.2

```
    byAccess : BYTE;        (* Früher Zugriffsberechtigung, kann von der Applikation des
                               Masters beliebig genutzt werden.*)
END_STRUCT
END_TYPE
```

An der Oberfläche verfügt CoDeSys über keinen Editor für dieses Objektverzeichnis.

Die EDS-Datei gibt nur vor, mit welchen Objekten das Objektverzeichnis angelegt wird. Dabei werden die Einträge immer mit der Länge 4 erzeugt und die Flags (niederstwertiges Byte der Komponente eines Objektverzeichniseintrags dwIdxSubIdxF) immer mit 16#41 bzw. 16#01 für den letzten Eintrag.

Wenn ein Objektverzeichnis im Master vorhanden ist, kann der Master als SDO-Server im Netz auftreten.

Immer wenn ein Client auf einen Objektverzeichniseintrag schreibend zugreift, wird das der Applikation über das Masterflag bMsgUsed angezeigt. Das ist eine Eingangsvariable des Masters, damit die Applikation das Flag zurücksetzen kann. Dieses wird immer vom Master immer nur gesetzt.

Die Applikation kann das Objektverzeichnis nutzen, indem die Einträge direkt beschrieben/gelesen werden, oder indem die Einträge auf IEC-Variablen zeigen. D.h. beim Lesen/Schreiben eines anderen Knoten wird direkt auf diese IEC-Variablen zugegriffen.

Ein Eintrag kann wie folgt angesprochen werden, wenn Index/Subindex bekannt sind:

```
I := FindInODMEntries(16#iiiiss00, pCanOpenMaster[0].wODMFirstIdx,
                      pCanOpenMaster[0].wODMFirstIdx + pCanOpenMaster[0]. wODMCount;)
```

Damit steht der Index des Eintrags in I zur Verfügung.

Nun kann direkt auf die Komponenten des Eintrags zugegriffen werden:
Um z.B. einen Eintrag direkt auf eine IEC Variable zeigen zu lassen, genügt es, Adresse, Länge und Flags einzutragen:

```
ODMEntries[I].dwContent := ADR(<Variablenname>);
ODMEntries[I].wLen := sizeof(<Variablenname>);
ODMEntries[I]. dwIdxSubIdxF := ODMEntries[I]. dwIdxSubIdxF OR OD_ENTRYFLG_WRITE
OR OD_ENTRYFLG_ISPOINTER;
```

Um nur den Inhalt des Eintrags zu ändern genügt es, dwContent zu verändern.

Alternativ kann man mit dem Index als Eingabeparameter auch die folgenden Funktionen verwenden:

GetODMEntryDataAddress, um auf die Adresse des Wertes des Eintrags zuzugreifen.

GetODMEntryDataLen, um die Länge der Daten des Eintrags zu ermitteln.

GetODMEntryValue, um den Wert des Eintrags abzufragen.

SetODMEntryValue, um den Wert des Eintrags zu setzen.

3 CanDevice

Eine CoDeSys-programmierbare Steuerung kann in einem CAN-Netzwerk auch als CANopen-Slave (auch CANopen-Node genannt, im folgenden als „CanDevice“ bezeichnet) erscheinen.

3.1 Funktionalität

Die CanDevice-Bibliothek zusammen mit dem CANopen-Konfigurator stellt dem Anwender folgende Funktionalität zur Verfügung:

- Konfiguration in CoDeSys der Eigenschaften NodeGuarding/Heartbeat, Emergency, NodeID und Baudrate, auf der das Device arbeiten soll.
- Zusammen mit dem Parameter-Manager in CoDeSys kann ein DefaultPDO-Mapping erstellt werden, das zur Laufzeit vom Master geändert werden kann. Die Änderung des PDO-Mappings erfolgt während der Konfigurationsphase durch den Master. Durch das Mapping können IEC-Variablen der Applikation in PDOs gemappt werden.
- Die CanDevice-Bibliothek stellt ein Objektverzeichnis zur Verfügung. Die Größe dieses Objektverzeichnisses wird zur Übersetzungszeit von CoDeSys festgelegt. In diesem Verzeichnis befinden sich alle Objekte, die das CanDevice beschreiben und zusätzlich die, die vom Parameter-Manager definiert sind. Im Parameter-Manager können zusammen mit dem CanDevice nur die Listenarten Parameter und Variablen verwendet werden.
- Die Bibliothek verwaltet die Zugriffe auf das Objektverzeichnis, tritt also am Bus als SDO-Server auf.
- Die Bibliothek überwacht das Nodeguarding bzw. Heartbeat-Consumer-Zeit (immer nur von einem Producer) und setzt entsprechende Fehlerflags für die Applikation.
- Erzeugung einer EDS-Datei, die die konfigurierten Eigenschaften des CanDevice so beschreibt, dass das Device als Slave unter einem CAN-Master konfiguriert werden kann.

Die CanDevice Bibliothek stellt ausdrücklich folgende, in CANopen beschriebenen Funktionalitäten nicht zur Verfügung. (Alle hier und im obigen Abschnitt nicht genannten Möglichkeiten des CANopen-Protokolls sind ebenfalls nicht implementiert) :

- Dynamische SDO, PDO-Identifier.
- Blockender SDO-Transfer.
- Implizite Erzeugung von Emergency-Nachrichten. Emergency Nachrichten müssen immer von der Applikation erzeugt werden. Die Bibliothek stellt dazu einen FB, der von der Applikation benutzt werden kann, zur Verfügung.
- Dynamische Änderungen der PDO-Eigenschaften/Laufzeiteigenschaften werden immer beim Eintreffen einer StartNode-NMT-Nachricht übernommen und mit jedem Zugriff auf ein Objekt in den Kommunikationseigenschaften des CanDevice.

3.2 CanDevice konfigurieren

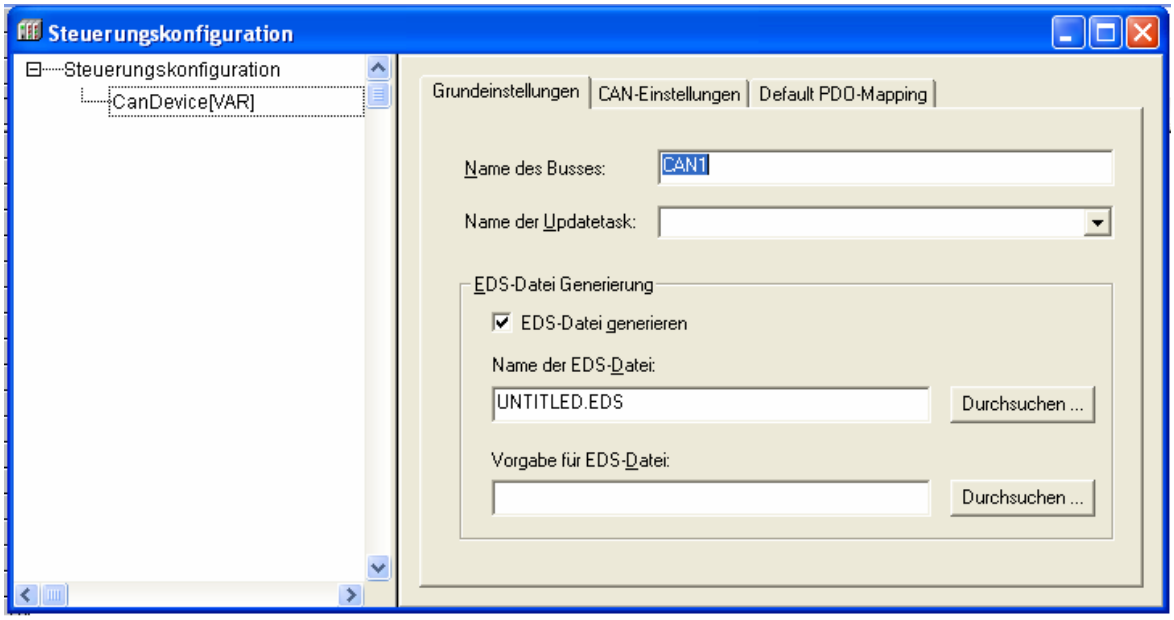
Um ein CanDevice zu konfigurieren, wählen Sie im Kontextmenu der Steuerungskonfiguration ‚Unterelement anhängen’ bzw. ‚Element einfügen’ -> ‚CanDevice’. Damit dieser Menüpunkt vorhanden ist, muss mindestens eine *.cfg-Datei, die die Beschreibung der Steuerungskonfiguration enthält (herstellerspezifisch), einen Eintrag für ein CanDevice enthalten:

```
[Module.CanDevice]
Name=CanDevice
Id=78379
DeviceType=CANDEVICE
BasisPrmDlg=FALSE
Class=<eine Klasse, die beim Root-Modul der Steuerungskonfiguration eingefügt werden kann>
```

tech_doc_d.dot / V1.2

Icon=codsmall.ico (Zeile ist optional)
FixedNumOfPDOs=1 (Zeile ist optional, das Default Mapping ist auf 4 PDOs fix eingestellt.)

Danach erhalten Sie folgenden Konfigurationsdialog:



3.3 CanDevice Einstellungen

Im Register Grundeinstellungen:

Name des Busses: wird im Moment nicht benutzt.

Name der Updatetask: Name der Task, in der der Aufruf des CanDevice erfolgt.

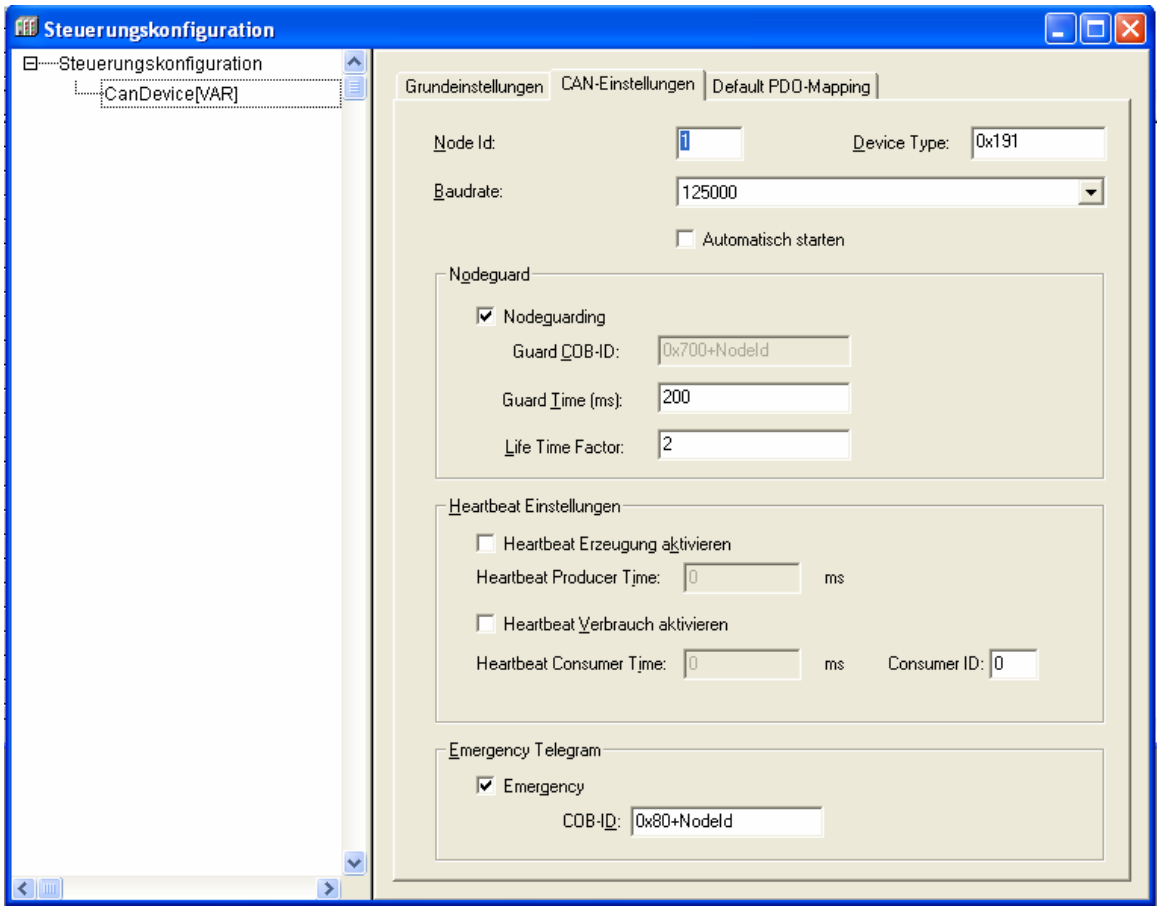
Soll aus den Einstellungen hier eine EDS-Datei erzeugt werden, um das CanDevice in eine beliebigen Masterkonfiguration einfügen zu können, muss hier die Option **EDS-Datei generieren** aktiviert werden und der **Name der EDS-Datei** angegeben werden. Optional kann auch noch eine Vorlagendatei angegeben werden (**Vorgabe für EDS-Datei**), deren Einträge zum EDS-File des CanDevice hinzugefügt werden. (Bei Überschneidungen werden Vorgaben der Vorlage nicht überschrieben.)

Im Register CAN Einstellungen:

Hier können die **NodeId** und die **Baudrate** eingestellt werden.

Der **Device Type** (der Defaultwert des Objekts 0x1000, der im EDS eingetragen wird) wird mit 0x191 (Standard IO Device) vorbelegt und kann vom Benutzer beliebig geändert werden.

Die **Nodeguard**-Parameter, die **Heartbeat**-Parameter und die **Emergency-COBID** werden hier festgelegt. Das Device kann hier nur für die Überwachung eines Heartbeats konfiguriert werden, nicht für eine Liste von Heartbeat Erzeugern. Normalerweise genügt es für einen Slave, den Heartbeat seines Masters zu überwachen.



Im Register „Default PDO-Mapping“:

Hier wir die Zuordnung zwischen lokalem Objektverzeichnis (Parameter-Manager) und den PDOs, die vom CanDevice gesendet/empfangen werden, festgelegt.

In den verwendeten Objektverzeichniseinträgen (Parameterliste vom Typ ‚Variablen’) wird zwischen Objektindex/Subindex die Verbindung zu Variablen der Applikation hergestellt. Dabei ist nur darauf zu achten, dass der Subindex 0 eines Indexes der mehr als einen Subindex enthält, implizit als Anzahl der Subindizes verwendet wird.

Beispiel:

> Ziel: Auf dem ersten Empfangs-PDO (COB-Id = 512 + NodeId) des CanDevice soll die Variable PLC_PRG.a empfangen werden.
>
> Also muss im Objektverzeichnis (Parameter-Manager) ein Index/SubIndex mit der Variablen PLC_PRG.a verknüpft werden. Dazu fügen Sie in einer Variablenliste eine Zeile hinzu, tragen Index und Subindex ein. Als Zugriff kommt nur „Write Only“ in Frage, da in ein Empfangs-PDO nur ein WriteOnly-Wert gemappt werden kann. In die Spalte Variable tragen Sie PLC_PRG.a ein, oder drücken F2 und wählen die Variable aus.

Hinweis: Um den Parameter-Manager öffnen zu können, muss in den Zielsystemeinstellungen unter „Netzfunktionen“ der Parameter-Manager aktiviert und mit sinnvollen Index/Subindexbereichen parametriert werden.

Im Default PDO-Mapping des CanDevice wird anschließend der Index/Subindexeintrag als Mapping-Eintrag einem Empfangs-PDO zugewiesen. Die Eigenschaften des PDOs lassen sich über den aus der Konfiguration der CAN-Slaves unterhalb einem Master bekannten Dialog festlegen.

Nur Objekte aus dem Parameter-Manager, die mit dem Attribut „read-only“ bzw. „write-only“ versehen sind, werden in der evtl. erzeugten EDS-Datei als mapbar markiert und tauchen in der Liste der

mapbaren Objekte auf. Die Objekte, die im Parametermanager angelegt werden, erhalten im EDS-File als DefaultValue entweder den Initialwert der Variablen (Variablenlisten) oder den angegebenen Wert (Parameterlisten).

Alle anderen Objekte werden in der EDS-Datei als nicht „mapbar“ markiert.

3.4 EDS-Datei erzeugen

Soll aus der Konfiguration des CanDevice eine EDS-Datei erzeugt werden, um das Device in der Konfiguration eines CANopen-Masters verwenden zu können, genügt es, im Register Grundeinstellungen die Funktionalität zu aktivieren. Zusätzlich muss der Dateiname der zu erzeugenden Datei angegeben werden. Die Datei wird beim Übersetzen des Projekts erzeugt.

Optional kann eine Vorlage für die EDS-Datei angegeben werden.

Alle vom Benutzer im Parameter Manager erzeugten Einträge (Objekte) erhalten in der EDS-Datei als Default-Wert den Initialwert der jeweiligen Variable (Variablenliste) oder den definierten Wert (Parameterliste)

Einträge, die in der Vorlagendatei vorhanden sind, werden von CoDeSys nicht geändert. Es ist also möglich, eigene Abschnitte in der EDS-Datei, die sich nicht ändern sollen, über eine Vorlagendatei zu definieren.

Hier die Inhalte der EDS Datei im einzelnen:

```
[FileInfo]
FileName=D:\CoDeSys\lib2\plcconf\MyTest.eds
FileVersion=1
FileRevision=1
Description=EDS for CoDeSys-Project:
D:\CoDeSys\CanOpenTestprojekte\TestHeartbeatODsettings_Device.pro
CreationTime=13:59
CreationDate=09-07-2005
CreatedBy=CoDeSys
ModificationTime=13:59
ModificationDate=09-07-2005
ModifiedBy=CoDeSys

[DeviceInfo]
VendorName=3S Smart Software Solutions GmbH
ProductName=TestHeartbeatODsettings_Device
ProductNumber=0x33535F44
ProductVersion=1
ProductRevision=1
OrderCode=xxxx.yyyy.zzzz
LMT_ManufacturerName=3S GmbH
LMT_ProductName=3S_Dev
BaudRate_10=1
BaudRate_20=1
BaudRate_50=1
BaudRate_100=1
BaudRate_125=1
BaudRate_250=1
BaudRate_500=1
BaudRate_800=1
BaudRate_1000=1
SimpleBootUpMaster=1
SimpleBootUpSlave=0
ExtendedBootUpMaster=1
ExtendedBootUpSlave=0

[DummyUsage]
Dummy0000=0
```

```
Dummy0001=0
Dummy0002=0
Dummy0003=0
Dummy0004=0
Dummy0005=0
Dummy0006=0
Dummy0007=0

[MandatoryObjects]
SupportedObjects=2
1=1000
2=1001

[1000]
SubNumber=0
ParameterName=Device Type
ObjectType=0x7
DataType=0x7
AccessType=ro
DefaultValue=0x191
PDOMapping=0

[1001]
SubNumber=0
ParameterName=Error Register
ObjectType=0x7
DataType=0x5
AccessType=ro
DefaultValue=
PDOMapping=0

[OptionalObjects]
SupportedObjects=8
1=0x1003
2=0x1005
3=0x100C
4=0x100D
5=0x1014
6=0x1016
7=0x1017
8=0x1018

[ManufacturerObjects]
SupportedObjects=5
1=0x3000
2=0x3001
3=0x3002
4=0x4000
5=0x5000

[1003]
SubNumber=2
ParameterName=Predefined error field

[1003sub0]
ParameterName=Number of entries
ObjectType=0x7
DataType=0x5
AccessType=rw
DefaultValue=1
PDOMapping=0

[1003sub1-5]
ParameterName=Predefined Error field
ObjectType=0x7
```

DataType=0x7
AccessType=rw
DefaultValue=0x0
PDOMapping=0

[1005]
SubNumber=0
ParameterName=COBID Synch Message
ObjectType=0x7
DataType=0x7
AccessType=rw
DefaultValue=0x80
PDOMapping=0

[100C]
SubNumber=0
ParameterName=Guard Time
ObjectType=0x7
DataType=0x6
AccessType=rw
DefaultValue=0
PDOMapping=0

[100D]
SubNumber=0
ParameterName=Lifetime Factor
ObjectType=0x7
DataType=0x5
AccessType=rw
DefaultValue=0
PDOMapping=0

[1014]
SubNumber=0
ParameterName=COBId Emergency
ObjectType=0x7
DataType=0x7
AccessType=rw
DefaultValue=0x80+$NodeId
PDOMapping=0

[1016sub0]
ParameterName=Nums consumer heartbeat time
ObjectType=0x7
DataType=0x5
AccessType=rw
DefaultValue=1
PDOMapping=0

[1016sub1]
ParameterName=Consumer heartbeat time
ObjectType=0x7
DataType=0x7
AccessType=rw
DefaultValue=0x0
PDOMapping=0

[1017]
SubNumber=0
ParameterName=Producer heartbeat time
ObjectType=0x7
DataType=0x6
AccessType=rw
DefaultValue=0
PDOMapping=0

```
[1018]
SubNumber=4
ParameterName=Vendor identification

[1018sub0]
ParameterName=Number of entries
ObjectType=0x7
DataType=0x5
AccessType=ro
DefaultValue=2
PDOMapping=0

[1018sub1]
ParameterName=VendorID
ObjectType=0x7
DataType=0x7
AccessType=ro
DefaultValue=0x0
PDOMapping=0

[1018sub2]
ParameterName=Product Code
ObjectType=0x7
DataType=0x7
AccessType=ro
DefaultValue=0x0
PDOMapping=0
```

Die Bedeutung der einzelnen Objekte entnehmen Sie der CANopen-Spezifikation DS301.

Die EDS-Datei enthält neben den obligatorischen Einträgen die Definitionen für SYNC, Guarding, Emergency und Heartbeat. Wenn diese Objekte nicht benutzt sind, wie hier im Beispiel Guarding und Heartbeat, sind die Defaultwerte zu 0 gesetzt. Die Objekte sind aber im Objektverzeichnis des Device zu Laufzeit vorhanden, also sind sie auch hier in der EDS-Datei beschrieben.

Analoges gilt für die (hier nicht aufgelisteten) Einträge für die Kommunikations- und Mapping-Parameter. Es sind immer alle 8 möglichen (Bitmapping wird von der Bibliothek nicht unterstützt!) Subindizes der Mapping-Objekte 0x16xx bzw. 0x1Axx vorhanden, aber u.U. im Subindex 0 nicht berücksichtigt.

Hinweis: Mit dem Eintrag FixedNumOfPDOs=1 in der cfg-Datei, in der Sektion des CanDevice's, werden die Mapping-Objekte, wenn z.B. gar kein Default-Mapping vorgegeben wurde, immer für 4 PDOs angelegt, so dass zur Laufzeit 4 PDOs zum „Mappen" von Variablen zur Verfügung stehen. Allerdings ist der Subindex 0 der Mapping-Objekte mit 0 vorbelegt, da in dem Fall, dass kein Default Mapping angegeben ist, keines der Objekte belegt ist.

3.5 Verändern des Standard-Mappings durch Master-Konfiguration

Das vorgegebene PDO-Mapping (in der CanDevice-Konfiguration) kann in bestimmten Grenzen durch den Master verändert werden.

Dabei gilt die Regel, dass das CanDevice nicht in der Lage ist, Objektverzeichniseinträge neu anzulegen, die nicht im Standard-Mapping (Default PDO-Mapping in der CanDevice-Konfiguration) vorhanden sind. Also kann z.B. für ein PDO, das im Default PDO-Mapping ein gemapptes Objekt enthält, in der Masterkonfiguration kein zweites Objekt gemappt werden.

Das durch die Masterkonfiguration veränderte Mapping kann also höchstens die im Standard-Mapping vorhandenen PDOs enthalten. Innerhalb dieser PDOs sind 8 Mapping-Einträge (Subindizes) vorhanden.

Eventuelle Fehler, die hierbei auftreten können, werden nicht angezeigt, sondern die überzähligen PDO-Definitionen / die überzähligen Mapping-Einträge werden so behandelt, als seien sie nicht vorhanden.

tech_doc_d.dot / V1.2

Die PDOs müssen im Master immer von 16#1400 (Empfangs-PDO-Kommunikationsparameter) bzw. 16#1800 (Sende-PDO-Kommunikationsparameter) beginnend angelegt sein und lückenlos aufeinander folgen.

Das Standard-Mapping des CanDevice ist immer nach dem Laden des Programms vorhanden. Es ist von der Masterkonfiguration abhängig, ob nicht vorhandene Mappings (in der Master-Konfiguration nicht vorhanden, jedoch in der Standardkonfiguration vorhanden) vom Master explizit überschrieben werden.

3.6 Arbeiten vom Anwenderprogramm mit dem CanDevice

3.6.1 Der Baustein CanopenDevice

Der Baustein wird für jedes konfigurierte CanDevice einmal instanziert. Die Komponenten des Funktkionsbausteins sind wie folgt verwendet:

bAutoStart : BOOL;	Speichert Option “Automatisch starten”
ucNodeId : BYTE;	Node-ID des CanDevice
wDrvNr : WORD;	Index des CAN-Controllers in der Konfiguration
wODStart : WORD;	Index des ersten Objektverzeichniseintrags dieses CanDevice.
wODEnd : WORD;	Der Index des letzten Objektverzeichniseintrags dieses CanDevice.
wPDOStartRx : WORD;	Index des ersten Rx-PDOs dieses CanDevice innerhalb des Arrays von RxPDOs (pCanDevPDO_Rx).
wPDOCountRx : WORD;	Anzahl der RxPDOs dieses CanDevice.
wPDOCountRxCurMap : WORD;	Laufvariable zur Konfiguration der PDO-Mappings aus dem Objektverzeichnis.
wPDOStartTx : WORD;	Index des ersten Tx-PDOs dieses CanDevice innerhalb des Arrays von TxPDOs (pCanDevPDO_Tx).
wPDOCountTx : WORD;	Anzahl der TxPDOs dieses CanDevice.
wPDOCountTxCurMap : WORD;	Laufvariable zur Konfiguration der PDO-Mappings aus dem Objektverzeichnis.
ucNodeIdNew : BYTE;	Diese Variable wird vom Initcode wie folgt gesetzt: Nach dem Aufruf der CAN-Treiber Funktion CanInit, in der der Treiber eine neue NodeID setzen kann, z.B. aus einem Drehschalter, wird dieser Variable der Wert: WORD_TO_BYTE(gCanInterface[1].pArray[1]) zugewiesen. CAN-Treiber müssen also eine neue NodeID in pArray[1] schreiben.
ErrorCode : WORD;	Wenn die Applikation eine Emergency Nachricht versenden will, wird hier der Übergabeparameter ErrorCode (nach CANopen) des FBs SEND_EMERGENCY eingetragen.
ErrorRegister : BYTE;	Wie ErrorCode.
ManufacturerErrorField : ARRAY[0..4] OF BYTE;	Analog zu ErrorCode.
bEmergencySent : BOOL;	Immer wenn eine Emergency-Nachricht gesendet wurde, wird diese Variable auf TRUE gesetzt.
bClearResetFlags : BOOL;	Die Applikation kann dieses Flag setzen, um die Flags, die NMT-Messages anzeigen zurückzusetzen.
bClearODEntryWritten : BOOL;	Die Applikation kann dieses Flag setzen, um die Flags, die einen Schreibzugriff auf das OD signalisieren zurückzusetzen.
bInit: BOOL;	Wird benutzt um den ersten Zyklus zu detektieren.
bMsgUsed : BOOL;	Interne Variable, die immer gesetzt wird, wenn eine Nachricht vom Device benutzt wurde.
bUseOldBootupToo : BOOL;	Dieses Flag kann von der Applikation im ersten Zyklus gesetzt werden, um das Device zu veranlassen, auch die alte Bootup-Message (Emergency mit 0 Byte Datenlänge) zu verschicken.

tech_doc_d.dot / V1.2

bGuardError: BOOL;	Wird von der Bibliothek gesetzt, wenn länger als GuardTime x LifeTimeFactor keine Guard-Message mehr angekommen ist.
ucState: BYTE := 127;	Aktueller Status des Device nach CANopen.
bHeartbeatError: BOOL;	Wenn das Device als Heartbeat-Consumer konfiguriert wurde, wird dieses Flag gesetzt, wenn länger als die konfigurierte Heartbeat-Consumertime keine neue Heartbeat-Message erkannt wurde.
bResetCommReceived : BOOL;	Reset Communication NMT Message wurde empfangen. Anzeige für die Applikation.
bResetNodeReceived : BOOL;	Reset Node NMT Message wurde empfangen. Anzeige für die Applikation.
bEnterPreOpReceived : BOOL;	Enter Preoperational NMT Message wurde empfangen. Anzeige für die Applikation.
bODEntryWritten: BOOL;	Ein Objektverzeichniseintrag wurde über einen SDO-Zugriff geschrieben.
iLastODEntryWritten: INT;	Index des zuletzt geschriebenen Eintrags.
i, iIdx: INT;	Interne Laufvariablen.
dwIdx: DWORD;	Interne Variable für den Objektverzeichniszugriff.
dwIdxHelp : DWORD;	Intern.
bSDOReadrqActive : BOOL := FALSE;	Interne Variable für SDO-Server Funktionalität.
bSDOWriterqActive : BOOL := FALSE;	Interne Variable für SDO-Server Funktionalität.
bSDOReadrspAbort : BOOL := FALSE;	Interne Variable für SDO-Server Funktionalität.
bSDOWriterspAbort : BOOL := FALSE;	Interne Variable für SDO-Server Funktionalität.
ucModus: BYTE;	Interne Variable für SDO-Server Funktionalität.
dwODEValue: DWORD;	Interne Variable für SDO-Server Funktionalität.
bSwap: BOOL;	Interne Variable, mit der bestimmt wird, ob das CanDevice auf einem Prozessor läuft, der Intel oder Motorola Byteorder benutzt. Die Reihenfolge der Bytes in einem SDOtransfer ist lt. CANopen immer Intel.
byAbortCode: ARRAY[0..3] OF BYTE;	Interne Variable für SDO-Server Funktionalität.
iIdxCom: INT;	Interne Variable
dwIdCom: DWORD;	Interne Variable
dwIdMap: DWORD;	Interne Variable
iIdxMap: INT;	Interne Variable
iPDOcur: INT;	Interne Variable
ucCurPDOLen : BYTE;	Interne Variable
iNumOfMappedObjects: INT;	Interne Variable
iMapObj: INT;	Interne Variable
iCurRxPDO: INT;	Interne Variable
iCurTxPDO: INT;	Interne Variable
bSendBootUp: BYTE;	Interne Variable, die immer dann zu TRUE wird, wenn die eine Bootup-Message verschickt werden soll.
bSendNodeGuard: BOOL;	Interne Variable zum Versenden der Nodeguard-Antwort.
ucToggle: BYTE;	Toggle-Bit der Nodeguard-Antwort.
bReentry : BOOL;	Historisch.
tGuardLifeTime : TON;	Der Überwachungstimer für das Nodeguarding.
bGuardingEnabled : BOOL;	Dieses Flag wird immer dann gesetzt, wenn die GuardLifeTime <> 0 ist.
tHeartbeatProducer : TON;	Heartbeatproducer-Zeit. In diesem Intervall werden Heartbeat-Messages erzeugt.

tech_doc_d.dot / V1.2

tHeartbeatConsumer : TON;	Die Zeit, mit der das Eintreffen der Heartbeat-Nachrichten überwacht wird.
byHeartbeatConsumerID : BYTE;	Die ID auf der die Heartbeat Nachrichten erwartet werden. Es wird immer nur eine ID überwacht.
bCurPDOChanged: BOOL;	Intern.
bTransmitCyclic: BOOL;	Intern
dwSynchCobId : DWORD;	Die COBID, auf der die Sync-Nachricht erwartet wird.
dwSynchWinLen : DWORD;	Konfigurierte SyncWindowLength.
dwSynComCycle: DWORD;	Konfigurierte Sync-Intervall.
diHelp : DINT;	Intern
dwHelp : DWORD;	Intern
MsgBuffer : CAN_Message;	Der Nachrichtenpuffer für empfangene Nachrichten.
a: INT;	Lebenszähler, der mit jedem Zyklus inkrementiert wird, um die Aufrufe beim Monitoring zu sehen.
dwCobIdPDOMin,dwCobIdPDOMax,dwAcceptance:DWORD;	Diese Werte werden dem Treiber übergeben, um ihm zu ermöglichen, eine Acceptancemask zu setzen.
bInitiateRspSend : BOOL;	Intern
bInitiateWrReqSend : BOOL;	Intern
iActiveSegSDORead:INT := -1;	Intern, für den segmentierten SDO-Transfer.
pActiveSegSDORead:POINTER TO BYTE;	Intern, für den segmentierten SDO-Transfer.
wSegSDOReadSendOffs : WORD;	Intern, für den segmentierten SDO-Transfer.
wSegSDOReadSendLen : WORD;	Intern, für den segmentierten SDO-Transfer.
iActiveSegSDOWrite:INT := -1;	Intern, für den segmentierten SDO-Transfer.
pActiveSegSDOWrite:POINTER TO BYTE;	Intern, für den segmentierten SDO-Transfer.
wSegSDOWriteRecvOffs : WORD;	Intern, für den segmentierten SDO-Transfer.
wSegSDOWriteRecvLen : WORD;	Intern, für den segmentierten SDO-Transfer.
wSwapCheck : WORD := 16#55AA;	Intern, muss so bleiben.
bErrCodeNot0: BOOL;	Check, ob der Treiber einen fatalen Fehler (BusOff) gesetzt hat.
dwEmergCobId : DWORD;	COBId, die zum Senden von Emergency Nachrichten benutzt wird.
iSDOWriteLen : INT;	Intern
iSDOReadLen : INT;	Intern
bGuardingStarted: BOOL;	Wird gesetzt, wenn die erste Guard-Nachricht eingetroffen ist.
bFirstHeartbeatRecv: BOOL;	Wird gesetzt, wenn die erste Heartbeat-Message eingetroffen ist.
bSub0NotFound: BOOL;	Wird intern benutzt, um den Abortcode „Index not found“ oder „SubIndex not found“ zu unterscheiden.

3.6.2 Zugriff auf die OD-Einträge vom Anwenderprogramm

Naturgemäß gibt es Objektverzeichniseinträge (OD-Einträge), die auf Variablen gemappt sind (Parameter-Manager).

Es gibt jedoch auch die implizit erzeugten Einträge des CanDevice, die nicht über den Parameter-Manager in einen Variableninhalt gemappt werden können. Diese Einträge sieht man erst zur Laufzeit des Systems im Array ODEntries. Im Monitorbetrieb (Online-Betrieb) von CoDeSys kann man alle Objektverzeichniseinträge in der globalen Variablenliste der 3s_CanOpenDevice.lib betrachten.

Der Zugriff auf die Inhalte dieser Objekte muss, genau wie Zugriffe auf die Parameterlisten des Parameter-Managers, applikationsgesteuert erfolgen:

Einmalig muss der Index des Eintrags ermittelt werden. Dazu schreibt man (hier für das erste CanDevice in der Konfiguration):

tech_doc_d.dot / V1.2

```
Index := FindBinary(16#iiiiss00, CanOpenDev[0].wODStart, CanOpenDev[0].wODEnd);
```

Anschließend kann mit GetODEntryValue(Index) der Inhalt des Objekts gelesen werden.

Mit SetODEntryValue kann der Inhalt des Objekts manipuliert werden. Alternativ kann auch direkt auf den Inhalt zugegriffen werden, wenn man, wie es für implizite Objekte immer der Fall ist, weiß dass es sich um einen numerischen Basisdatentyp handelt:

```
ODEntries[Index].dwContent := xxxxx;
```

3.6.3 Ändern der PDO-Eigenschaften zur Laufzeit

Sollen die Eigenschaften eines PDOs zur Laufzeit verändert werden, so funktioniert das durch einen anderen Knoten über SDO-Schreibzugriffe, wie dies von CANopen beschrieben wird. Alternativ kann man auch direkt eine neue Eigenschaft, wie z.B. die Eventtime eines Sende-PDO's schreiben und anschließend einen StartNode-NMT-Befehl an den Knoten schicken, obwohl er bereits gestartet ist. Das führt dazu, dass das Device die Werte im Objektverzeichnis neu interpretiert.

Aber auch die lokale Applikation kann natürlich die Eigenschaften eines PDOs verändern.

Dazu muss die Applikation lediglich wie oben beschrieben die Inhalte des/der betreffenden Objekte verändern und anschließend die Methode SetupPDOTable des CanDevice aufrufen:

```
CanOpenDev[0].SetupPDOTable();
```

3.6.4 Emergency-Nachrichten durch das Anwenderprogramm senden

Um eine Emergency-Nachricht durch das Anwenderprogramm zu versenden, kann der Baustein (FB) SEND_EMERGENCY benutzt werden.

Der Baustein hat die folgende Schnittstelle:

```
VAR_INPUT
    ENABLE : BOOL;          (* Solange Enable ansteht, wird mit jedem Aufruf eine Emergency-
                               Nachricht verschickt.*)
    ErrorCode : WORD;       (* Der Error-Code, nach CANopen sind einige Werte bereits
                               vorbestimmt.
                               (Siehe dazu die Tabelle unten.)*)
    ErrorRegister : BYTE;   (* Das Error-Register ist der Inhalt des Objekts 1001h.*)
    ManufacturerErrorField : ARRAY[0..4] OF BYTE; (* Hier kann die Applikation ihre eigenen
                                                     Fehlercodes definieren.*)
    DeviceIndex : INT;      (*Der 0-basierte Index des Device, das die Emergency senden
                               soll.*)
END_VAR
VAR_OUTPUT
    bReady : BOOL;          (*Wenn die Nachricht gesendet wurde, geht bReady auf TRUE.*)
END_VAR
```

Die Tabelle mit ErrorCodes aus der DSP301:

```
00xx Error Reset or No Error
10xx Generic Error
20xx Current
21xx Current, device input side
22xx Current inside the device
23xx Current, device output side
30xx Voltage
31xx Mains Voltage
32xx Voltage inside the device
33xx Output Voltage
40xx Temperature
41xx Ambient Temperature
42xx Device Temperature
50xx Device Hardware
60xx Device Software
```

61xx Internal Software
62xx User Software
63xx Data Set
70xx Additional Modules
80xx Monitoring
81xx Communication
8110 CAN Overrun (Objects lost)
8120 CAN in Error Passive Mode
8130 Life Guard Error or Heartbeat Error
8140 recovered from bus off
8150 Transmit COB-ID
82xx Protocol Error
8210 PDO not processed due to length error
8220 PDO length exceeded
90xx External Error
F0xx Additional Functions
FFxx Device specific

© 3S - Smart Software Solutions GmbH

4 CAN Netzwerkvariablen

Netzwerkvariablen sind eine Möglichkeit, Daten zwischen zwei oder mehreren Steuerungen auszutauschen. Der Mechanismus sollte dabei für den Anwender möglichst einfach zu handhaben sein. Derzeit sind Netzwerkvariablen auf Basis von CAN und UDP implementiert. Die Variablenwerte werden dabei auf der Basis von Broadcast-Nachrichten automatisch ausgetauscht. In UDP sind diese als Broadcast-Telegramme, in CAN als PDOs realisiert. Diese Dienste sind vom Protokoll her nicht bestätigte Dienste, d.h. es gibt keine Kontrolle, ob die Nachricht auch beim Empfänger ankommt. Netzwerkvariablenaustausch entspricht einer 1 (Sender) zu n (Empfänger) – Verbindung.

Der **Parameter-Manager** ist eine weitere Möglichkeit, Variablen auszutauschen. Dabei handelt es sich um eine 1 zu 1 – Verbindung, die ein bestätigtes Protokoll verwendet. Hier kann der Anwender also kontrollieren, ob die Nachricht den Empfänger erreichte. Der Austausch erfolgt nicht automatisch, sondern über den Aufruf von Funktionsblöcken aus dem Anwenderprogramm.

4.1 CAN-Netzwerk-Variablen für den Anwender

Um die Netzwerkvariablen mit CoDeSys zu nutzen, sind für den Anwender folgende Schritte nötig:

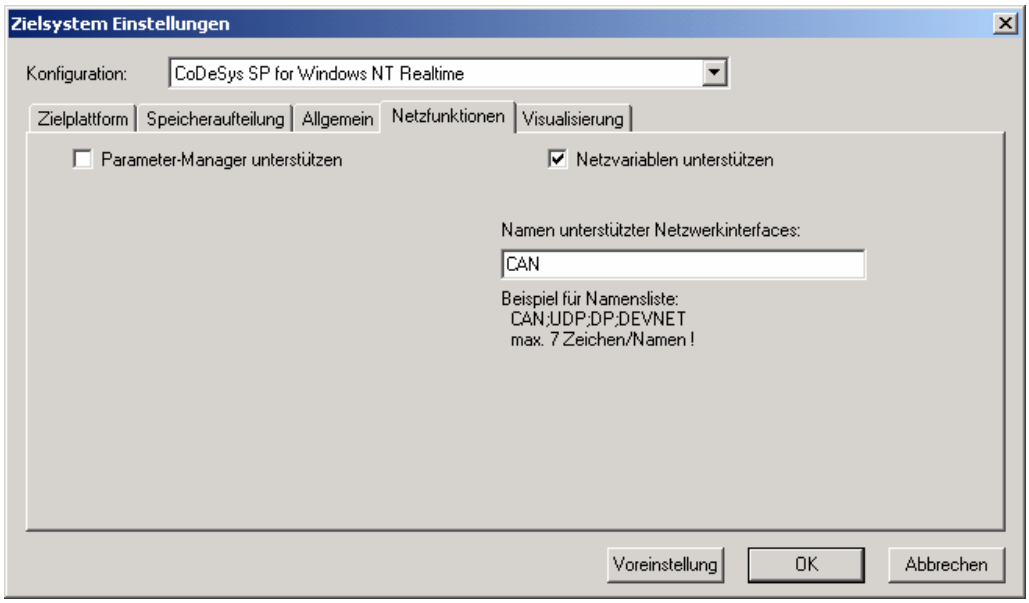
4.1.1 Voraussetzungen

Benötigt wird ein CoDeSys Programmiersystem Version 2.3 SP5 (oder neuer) und die Bibliotheken

3s_CanDrv.lib, 3S_CanOpenManager.lib und 3S_CanOpenNetVar.lib

CoDeSys erzeugt automatisch den nötigen Initialisierungscode und den Aufruf der Netzwerk-Bausteine am Zyklusanfang und -ende.

4.1.2 Einstellungen in den Zielsystemeinstellungen

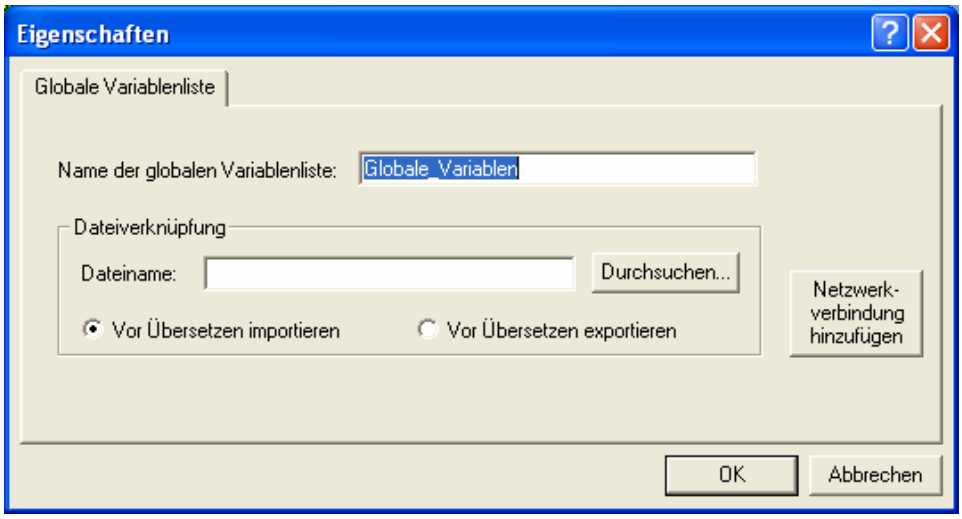


Aktivieren Sie das Kontrollkästchen **Netzvariablen unterstützen** im Tab 'Netzfunktionen' der Dialogbox 'Zielsystemeinstellungen'. Bei **Namen unterstützter Netzwerkinterfaces** geben Sie den Namen des gewünschten Netzwerks an, hier CAN.

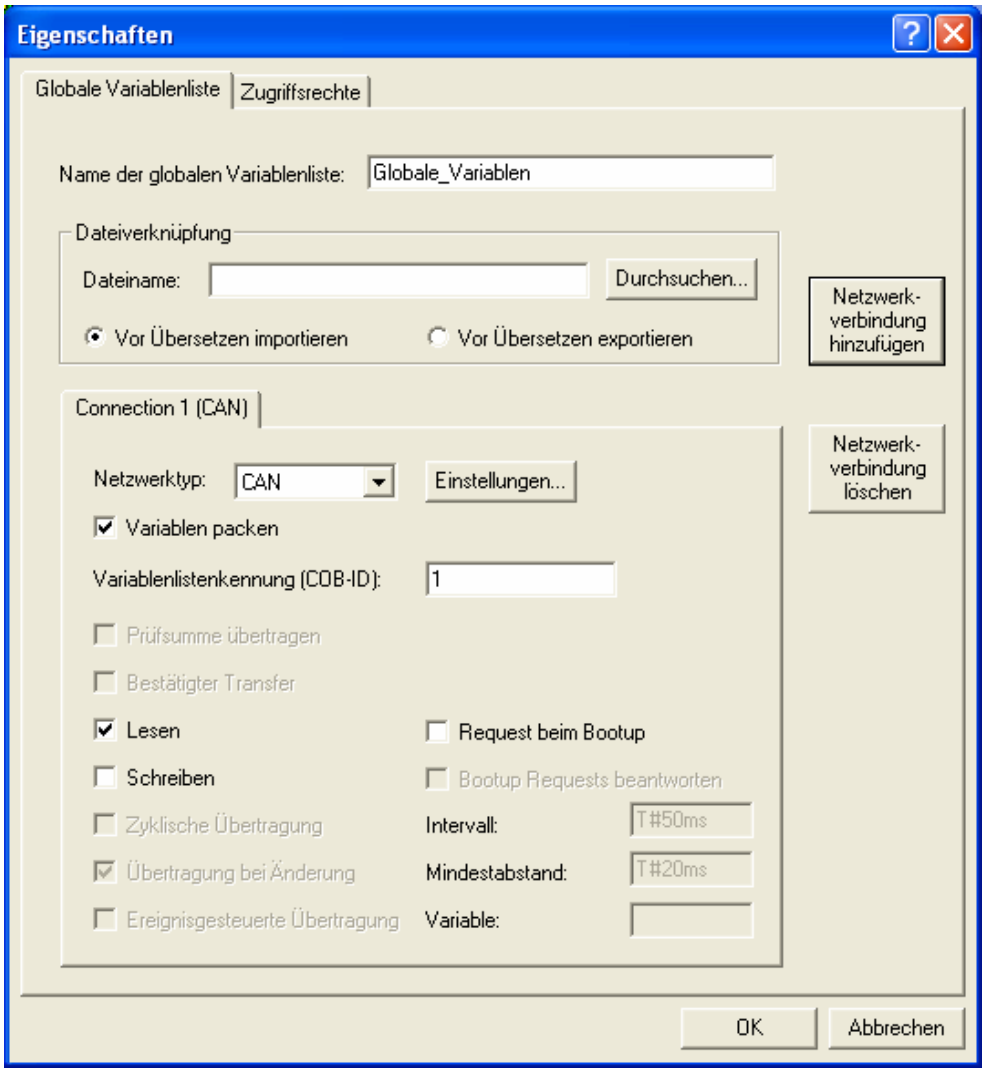
4.1.3 Einstellungen in den globalen Variablenlisten

Legen Sie eine neue globale Variablenliste an. Hier definieren Sie die Variablen, die sie mit anderen Steuerungen austauschen wollen. Folgender Dialog öffnet automatisch (und kann später auch über den Befehl Befehl **'Objekt Eigenschaften'** geöffnet werden, wenn die globale Variablenliste im Object Organizer selektiert ist):

tech_doc_d.dot / V1.2



Wenn sie die Schaltfläche **Netzwerkverbindung hinzufügen** drücken, können Sie die Netzwerkeigenschaften dieser Variablenliste definieren. Wenn Sie mehrere Netzwerkverbindungen konfiguriert haben, können Sie hier auch pro Variablenliste mehrere Verbindungen konfigurieren.



Die Optionen haben dabei folgende Bedeutung:

Netzwerktyp: Hier kann einer der bei den Zielsystemeinstellungen angegebenen Netzwerknamen eingetragen werden.

Lesen: Die Variablenwerte werden von einer (oder mehreren) anderen Steuerungen gelesen.

Schreiben: Die Variablen dieser Variablenliste werden zu anderen Steuerungen gesendet. Es wird empfohlen, für eine Variablenliste nur eine dieser Möglichkeiten zu wählen, also entweder nur lesen oder nur schreiben. Sollen verschiedene Variablen eines Projekts gelesen und geschrieben werden, so verwenden Sie mehrere Variablenlisten: Eine zum Lesen, eine zum Schreiben. Außerdem wird empfohlen, dass in einem Netzwerk nur eine Steuerung die gleiche Variablenliste sendet.

Zyklische Übertragung: Nur gültig, wenn „Schreiben“ aktiviert. Die Werte werden in der angegebenen Intervallzeit gesendet, unabhängig davon, ob sie sich geändert haben.

Übertragung bei Änderung: Die Werte werden nur gesendet, wenn sich der Wert geändert hat, oder wenn die bei Mindestintervall angegebene Zeit abgelaufen ist.

Variablen packen: Wenn diese Option aktiviert ist, werden die Variablen nach Möglichkeit in einer Übertragungseinheit zusammengefasst. Bei UDP ist eine Übertragungseinheit 256 Bytes groß. Bei CAN ist eine Übertragungseinheit 8 Bytes groß. Passen nicht alle Variablen der Liste in eine Übertragungseinheit, dann werden für diese Liste mehrere Übertragungseinheiten gebildet. Ist die Option nicht aktiviert, kommt jede Variable in eine eigene Übertragungseinheit. Wenn Übertragung bei Änderung konfiguriert ist, wird für jede Übertragungseinheit getrennt geprüft, ob sie geändert ist und gesendet werden muss.

Variablenlistenkennung (COBID): Diese ID wird als eindeutige Kennung benutzt, um Variablenlisten verschiedener Projekte auszutauschen. Nur Variablenlisten mit gleicher Variablenlistenkennung können Daten austauschen. Es ist darauf zu achten, dass die Definitionen der Variablenlisten mit gleichem Basisidentifier in den verschiedenen Projekten übereinstimmen. Hierzu kann das Feature Dateiverknüpfung verwendet werden: Aus einem Projekt wird die Variablenliste exportiert, die anderen Projekte importieren sie, wodurch der Inhalt nicht neu eingegeben werden muss.
WARNUNG: Die Variablenlistenkennung wird in CAN-Netzwerken direkt als COBID der CAN-Nachrichten benutzt. Es gibt keine Überprüfung, ob der Identifier auch in der CAN-Konfiguration für IO-PDOs benutzt wird!

Damit die Daten korrekt zwischen zwei Steuerungen ausgetauscht werden ist es erforderlich, dass die globalen Variablenlisten in den beiden Projekten übereinstimmen. Sie können das Feature „**Dateiverknüpfung**“ benutzen, um dies sicherzustellen. Ein Projekt kann die Datei vor dem Übersetzen exportieren, die anderen Projekte sollten die vor dem Übersetzen importieren.

Neben einfachen Datentypen kann eine Variablenliste auch Strukturen und Arrays enthalten. Die Elemente dieser zusammengesetzten Datentypen werden einzeln versendet.

Wenn eine Variablenliste größer als ein PDO des entsprechenden Netzwerks ist, werden die Daten auf mehrere PDOs aufgeteilt. Es kann darum nicht zugesichert werden, dass alle Daten der Variablenliste in einem Zyklus empfangen werden. Teile der Variablenliste können in verschiedenen Zyklen empfangen werden. Dies ist auch für Variablen mit Struktur- und Arraytypen möglich.

Für den CAN nicht unterstützte Optionen sind:

- Prüfsumme übertragen.
- Bootuprequests beantworten.
- Bestätigter Transfer.
- Request beim Bootup.

4.1.4 Erzeugte Aufrufe

Es wird eine implizite Variablenliste erzeugt. Für jedes Sende- und Empfangs-PDO wird ein Arrayeintrag vom Typ NetVarPDO_Rx_CAN bzw. NetVarPDO_Tx_CAN erzeugt.

Für Debug-Zwecke wird eine Datei mit den generierten Deklarationen erzeugt. Die Datei heißt NetworkGlobalVars_CAN.exp und ist im Compile-Verzeichnis zu finden. (Nur wenn CoDeSys mit der Kommandozeilenoption /debug gestartet wurde.)

Es wird Initialisierungscode erzeugt, der die erzeugten Daten initialisiert. Dieser wird aus dem GlobalInit-Baustein aufgerufen, der wiederum zum Initialisieren der Projektdaten direkt nach dem Download aufgerufen wird.

Am Beginn und am Ende einer Task werden implizite Aufrufe von Bibliotheksbausteinen erzeugt:

Für jede Task, in der Netzwerkvariablen verwendet werden,

- wird zu Beginn für jedes PDO ein Empfangs-Aufruf erzeugt.
- wird zu Beginn ein Aufruf der (einzigen) Instanz des Bausteins NetVarManager_Udp_FB bzw. NetVarManager_Can_FB erzeugt:

```
NetVarManager_UDP();
```

- wird am Ende für jedes PDO ein Sende-Aufruf erzeugt.

Die Aufrufe insgesamt:

```
CAN_Read(0);
NetVarManager_CAN();
pNetVarPDO_Rx_CAN[0]();
<Applikationscode>
MgrClearRxBuffer(wCurTask:= 1,wDrvNr := 0, dwFlags := 0, dwPara := 0);
pNetVarPDO_Tx_CAN[0]();
```

5 Anhang

5.1 Liste der normalerweise für einen Slave erzeugten SDOs

Der CANopen-Konfigurator von CoDeSys erzeugt standardmäßig folgende SDOs:

- Lesezugriff auf das Objekt 0x1000, um den Typ des Nodes zu überprüfen.
- Schreibzugriff auf alle Objekte des EDS-Files, die in der aktuellen Konfiguration des Slaves nicht dem Defaultwert im EDS-File entsprechen und im EDS-File mit Schreibzugriffsrecht versehen sind.
- Wenn in der aktuellen Konfiguration der Schalter „Alle SDOs erzeugen“ gesetzt ist, werden alle Kommunikations- und Mapping-Parameter (1400er, 1600er, 1800er und 1A00er Objekte) immer erzeugt, die ServiceData-Objekte (eigener Tab im Konfigurator) nach wie vor nur bei Änderung gegenüber dem EDS-File.

5.2 Erkennen eines Schreibpufferüberlaufs

In der globalen Variablenliste der 3S_CanOpenManager.lib existiert ein Array von Flags, das den Überlauf des Schreibpuffers eines CanControllers anzeigt:

g_CanMgrTxBufferOverrun[<Controllerindex>].

Die Applikation kann diese Flags zu Meldungs-/Diagnosezwecken benutzen. Die Flags müssen von der Applikation rückgesetzt werden, die Bibliothek setzt diese nur.

5.3 FAQs – Häufig gestellte Fragen

Sehen Sie im folgenden einige häufig gestellte Fragen zum Thema CAN und CoDeSys und die Antworten dazu. Beachten Sie auch die mit dem vorliegenden Dokument gelieferten Beispieldateien und –projekte.

HINWEIS: Wenn Sie Anfragen zum Thema CANopen an den 3S-Support richten, fügen Sie bitte immer das betreffende Projekt, EDS-Datei(en) und ggfs. die Aufzeichnung von CAN-Meldungen hinzu.

5.3.1 CANopen Funktionalität

F1: Zur Bildung eines CANopen Slaves: Ist es richtig, dass über den Parameter Manager die Objektliste gefüllt wird?

A: Ja.

F2: Kann man die Objektliste auch im CoDeSys Code generieren/erweitern und wie?

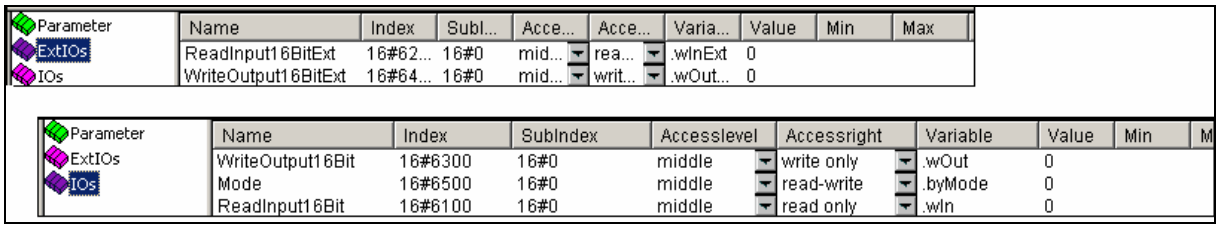
A: Die Objektliste für den Listentyp *Variablen* kann mittels Pragmas auch aus dem Code heraus generiert werden. (Zur Verwendung von Pragmas sehen Sie auch die Benutzerdokumentation bzw. Online Hilfe zu CoDeSys).
Sehen Sie im folgenden ein Beispiel, in dem in einer globalen Variablenliste Pragmas eingefügt sind, die bewirken, dass die betreffenden Variablen im Parameter Manager in einer Objektliste angelegt werden. (Die entsprechende Konfiguration der Indexbereiche muss in den CoDeSys Zielsystemeinstellungen in Kategorie „Netzfunktionen“ vorliegen).

```
VAR_GLOBAL
  g_byCANNodeID : BYTE := 16#10; (* CANopen NodeID normally read from a rotary switch. *)
(*
 Format of the ParameterManager entry:
  {parameter list=<name> [ name= index=16 subindex= accesslevel= accessright= value= ] }
 Possible keys and its values:
  name              : The name is only for information purposes
                      (not shown in PLC configuration or EDS file).
  index             : Index of the variable.
  subindex          : SubIndex of the variable.
  accesslevel : Not used for CAN, but must be defined.
              - low    :
              - middle :
              - high   :
  accessright : Accessright values:
              - "read only"  : Tx PDO (Input, Master's view)
              - "write only" : Rx PDO (Output, Master's view)
  value             : Default value.
*)

  (* Objects mapped by default *)
  wIn     : WORD {parameter list=IOs    [name=ReadInput16Bit  index=16#6100 subindex=16#0
accesslevel=middle accessright="read only"  value=0 ] };
  wOut    : WORD {parameter list=IOs    [name=WriteOutput16Bit   index=16#6300 subindex=16#0
accesslevel=middle accessright="write only" value=0 ] };
  (* Not mappable *)
  byMode  : BYTE {parameter list=IOs    [name=Mode    index=16#6500 subindex=16#0
accesslevel=middle accessright="read-write" value=0 ] };

  (* Objects could be mapped, if required. *)
  wInExt  : WORD {parameter list=ExtIOs [name=ReadInput16BitExt  index=16#6200 subindex=16#0
accesslevel=middle accessright="read only"  value=0 ] };
  wOutExt : WORD {parameter list=ExtIOs [name=WriteOutput16BitExt  index=16#6400 subindex=16#0
accesslevel=middle accessright="write only" value=0 ] };
END_VAR
```

-> Erzeugte Listen im Parameter Manager:



Parameter	Name	Index	Subl...	Acce...	Acce...	Varia...	Value	Min	Max
ExtIOs	ReadInput16BitExt	16#62...	16#0	mid...	rea...	.wInExt	0		
IOs	WriteOutput16BitExt	16#64...	16#0	mid...	writ...	.wOut...	0		

Parameter	Name	Index	SubIndex	Accesslevel	Accessright	Variable	Value	Min	M
ExtIOs	WriteOutput16Bit	16#6300	16#0	middle	write only	.wOut	0		
IOs	Mode	16#6500	16#0	middle	read-write	.byMode	0		
	ReadInput16Bit	16#6100	16#0	middle	read only	.wIn	0		

F3: Funktionieren die NMT-Befehle: Boot-Up, Umschaltung des Mode (Operational/Preoperational), Reset, Emergency, Node-Guarding?

A: Ja, diese Befehle funktionieren. Wird ein `ResetNode` oder `ResetCommunication` empfangen, dann werden die Werte der Objekte normalerweise auf Ihren Default-Wert

zurückgesetzt. Das muss im Fall des CanDevice die Applikation übernehmen. Die Applikation kann anhand der beiden Variablen: `bResetCommReceived` und `bResetNodeReceived` feststellen, ob einer der beiden NMT-Services empfangen wurde.

F4: Kann ich Guard-Time und Life-Time-Faktor auch über mein Objektverzeichnis ändern?

A: Ja. Wenn über einen SDO-Zugriff darauf geschrieben wird, ändert die Bibliothek von sich aus die entsprechenden Variablen. Wenn Sie von der Applikation aus ins OD schreiben, müssen Sie, wenn die Änderungen eine Wirkung haben sollen,

```
CanOpenDev[0].SetupPDOTable();
```

aufrufen. Diese Methode des CanDevice bewirkt, dass das gesamte OD neu interpretiert wird.

F5: Können Node-Nr. und Baudrate auch über den CoDeSys Code eingestellt werden?

A: Wenn Sie die Variable `nNewNodeId` setzen und dann die Methode

```
Init();
```

aufrufen, bekommt das Device eine neue NodeID, alle NodeID-relativen Werte im OD werden angepasst.
Wenn Sie die Baudrate ändern wollen, müssen Sie diese in die Struktur `g_CanInterface[xx]` eintragen und danach

```
Can_Init(xx,ADR(g_CanInterface[xx]));
```

aufrufen. Damit wird dem Treiber die Änderung mitgeteilt.

F6: Bei den „Netzfunktionen“ der Zielsystemeinstellungen wird die Objektverzeichnisgröße eingestellt. Bezieht sich dies nur auf den selbstdefinierten Bereich (z.B. 2000...) oder auch auf den Standardbereich (1000...)?

A: Nur die im Parameter Manager definierten Objekte müssen hier angegeben werden.

F7: Mein kundenspezifisches Objektverzeichnis fängt bei 2000 an und geht bis zum Standard I/O-Bereich (6400). Ist diese Größe ein Problem?

A: Nein, solange die verwendete Steuerung genug Speicher für diese Variablen hat. Wenn Sie ein Zielsystem mit einem C16x-Controller benutzen (65-K-Segment-Grenze) dann kann eine Variable nur 64-K groß werden. `ODEntries` ist ein Array von Strukturen, jede braucht 12 Byte, also kann man theoretisch maximal 65536 / 12 = 5461 Einträge anlegen. Da aber auf einem C16x-System der Baustein, der alle Variablen zusammen initialisiert, maximal ebenfalls nur 64K groß werden kann, liegt diese Zahl wesentlich tiefer, etwa bei 500 bis 1000, das ist stark abhängig vom Rest der Applikation.

F8: Ab Index 4000,0 benötige ich 130 Einträge mit Strings mit jeweils 32 Zeichen. Ist dies ein Problem? Wie bekomme ich die Strings hier in das Objektverzeichnis?

A: Wie alle anderen Variablen auch: Die Strings müssen IEC-Variablen sein, dann kann man sie genauso wie numerische Variablen über den Parameter-Manager einhängen und über SDO-Transfers darauf zugreifen.

F9: Kann die Mapping-Tabelle von außen (sprich vom Master) bei mir (sprich in meinem Objektverzeichnis) beschrieben werden?

A: *siehe Kap. 3.5 Verändern des Standard-Mappings durch Master-Konfiguration*
Das vorgegebene PDO-Mapping (in der CanDevice-Konfiguration) kann in bestimmten Grenzen durch den Master verändert werden.
Dabei gilt die Regel, dass das CanDevice nicht in der Lage ist, Objektverzeichniseinträge neu anzulegen, die nicht im Standard-Mapping (Default PDO-Mapping in der CanDevice-Konfiguration) vorhanden sind. Also kann z.B. für ein PDO, das im Default PDO-Mapping ein gemapptes Objekt enthält, in der Masterkonfiguration kein zweites Objekt gemappt werden.

Das durch die Masterkonfiguration veränderte Mapping kann also höchstens die im Standard-Mapping vorhandenen PDOs enthalten. Innerhalb dieser PDOs sind 8 Mapping-Einträge (Subindizes) vorhanden.
Eventuelle Fehler, die hierbei auftreten können, werden nicht angezeigt, sondern die überzähligen PDO-Definitionen / die überzähligen Mapping-Einträge werden so behandelt, als seien sie nicht vorhanden.
Die PDOs müssen im Master immer von 16#1400 (Empfangs-PDO-Kommunikationsparameter) bzw. 16#1800 (Sende-PDO-Kommunikationsparameter) beginnend angelegt sein und lückenlos aufeinander folgen.
Das Standard-Mapping des CanDevice ist immer nach dem Laden des Programms vorhanden. Es hängt von der Masterkonfiguration ab, ob nicht vorhandene Mappings (in der Master-Konfiguration nicht vorhanden, jedoch in der Standardkonfiguration vorhanden) vom Master explizit überschrieben werden.

F10: Wie viele PDOs sind möglich?

A: Es gibt hier nur eine Speichergrenze, keine systematische.

F11: Index 1000,1008,1009,100A muss ich mit meinen gerätespezifischen Daten beschreiben. Wie ist dies möglich?

A: *siehe Kap. 3.6.2 Zugriff auf die OD-Einträge vom Anwenderprogramm*
Naturgemäß gibt es Objektverzeichniseinträge (OD-Einträge), die auf Variablen gemappt sind (Parameter-Manager).
Es gibt jedoch auch die implizit erzeugten Einträge des CanDevice, die nicht über den Parameter-Manager in einen Variableninhalt gemappt werden können. Diese Einträge sieht man erst zur Laufzeit des Systems im Array `ODEntries`. Im Monitorbetrieb (Online-Betrieb) von CoDeSys kann man alle Objektverzeichniseinträge in der globalen Variablenliste der *3s_CanOpenDevice.lib* betrachten.
Der Zugriff auf die Inhalte dieser Objekte muss, genau wie Zugriffe auf die Parameterlisten des Parameter-Managers, applikationsgesteuert erfolgen:
Einmalig muss der Index des Eintrags ermittelt werden. Dazu schreibt man (hier für das erste CanDevice in der Konfiguration):

```
Index    :=    FindBinary(16#iiiiss00,    CanOpenDev[0].wODStart,
CanOpenDev[0].wODEnd);
```

Anschließend kann mit `GetODEntryValue(Index)` der Inhalt des Objekts gelesen werden.

Mit `SetODEntryValue` kann der Inhalt des Objekts manipuliert werden. Alternativ kann auch direkt auf den Inhalt zugegriffen werden, wenn man, wie es für implizite Objekte immer der Fall ist, weiß dass es sich um einen numerischen Basisdatentyp handelt:

```
ODEntries[Index].dwContent := xxxxx;
```

F12: Wenn von außen auf mein Objektverzeichnis geschrieben wird, erkenne ich dies nur durch die Änderung der entsprechenden Variablen, welche dem entsprechenden Index/Subindex zugeordnet ist. Gibt es auch eine Möglichkeit im CoDeSys Code, einkommende Schreibzugriffe auf mein Objektverzeichnis direkt zu empfangen (mit CoDeSys Anweisung?)

A: siehe Kap. 3.6.1: Beschreibung der Schnittstelle des Bausteins CANopenDevice: Variable `bODEntryWritten und iLastODEntryWritten.`

F13: Es können nur read-only und write-only Objekte verarbeitet werden. Bei meiner Anwendung muss ich definitiv auch read-write Objekte verwenden. Gibt es vielleicht schon ein Update, mit dem dieses Problem behoben ist?

tech_doc_d.dot / V1.2

A: Read-Write Objekte werden beim Master im Reiter *Service Data Objects* (-> ErrorModeOutput16Bit) angezeigt. Diese können dort auch verändert werden. Werden diese Werte geändert (Abweichung vom Standard-Wert), so wird das Objekt in der Startup-Phase an das CANDevice gesendet.
Es ist allerdings nicht möglich, diese Objekte in PDOs zu mappen. Lt. CANopen sind mapbare Objekte immer entweder read-only oder write-only.

F14: Mich stört die Boot-Up Message nach V4; ich benötige die V3 Variante.

A: Sie können eine V3 Bootup Message benutzen, zusätzlich zu der anderen, es werden dann beide geschickt, indem Sie im ersten SPS-Zyklus die Variable

```
bUseOldBootupToo := TRUE;
```

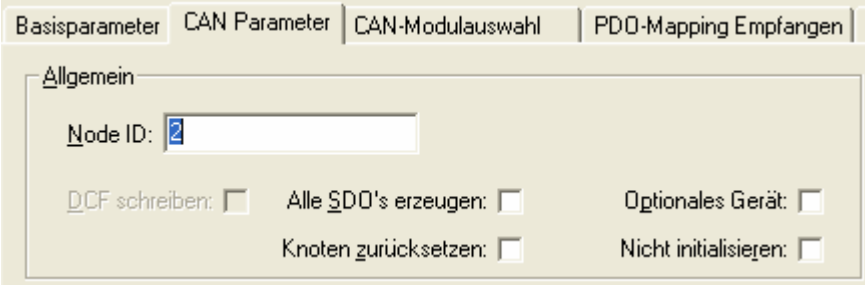
setzen.

5.3.2 CANopen Master

5.3.2.1 Konfiguration

In diesem Abschnitt finden Sie alle Fragen zur Konfiguration des CANopen-Masters.

F1: Wie können die CAN-Parameter-Optionen Optionales Gerät und Nicht initialisieren in der Steuerungskonfiguration eingeblendet werden?



A: Die CAN-Parameter Optionen **Optionales Gerät** und **Nicht initialisieren** können über die Steuerungskonfigurationsdatei aktiviert werden. Dazu müssen Sie in der Steurerungskonfigurationsdatei beim CANopen-Slave die Einträge **OptionalDevice** und **NoInitialize** eintragen.

```
[Module.CanSlave]
Name=CanSlave
Id=693734
…
OptionalDevice=TRUE
NoInitialize=TRUE
```

Der reine Eintrag bewirkt, dass die Optionen in der Steuerungskonfiguration eingeblendet werden. Mit FALSE oder TRUE können Sie dann den Default-Wert festlegen.

F2: Im Dialog CAN Parameter der Steuerungskonfiguration gibt es die Option Knoten zurücksetzen. Diese Option ist bei mir deaktiviert und grau.

Nun meine Fragen:

- **Gehe ich recht in der Annahme, dass ich mit dieser Option einstellen kann, ob an den Knoten vor der Konfiguration ein NMT Reset Node Telegramm gesendet wird (andernfalls NMT Reset Communication) ?**
- **Wie kann ich diese Option anklickbar machen?**
- **Wie kann ich einen Vorgabewert für diese Option setzen (gesetzt / nicht gesetzt)?**
- **Ist der Wert der Option irgendwo in der Download-Struktur verfügbar? (die Struktur CCCanSlave in RTSCFG.H enthält jedenfalls keinen entsprechenden Eintrag).**

A: Diese Option wird aktiviert, wenn das Objekt 0x1011 in der EDS-Datei steht, bzw. deaktiviert, wenn dieses Objekt nicht in der Datei vorhanden ist. Einen Default-Wert kann man dort nicht setzen.
Wird ein SDOWrite auf diesem Objekt ausgeführt, dann werden die Default-Werte zurückgespeichert und ein ResetNode ausgeführt.

In der Download-Struktur gibt es keine direkte Variable, die diesen Wert angibt. Sie können aber in der Liste der SDOs nachsehen, ob dieses Objekt vorhanden ist. Ist dieses Objekt vorhanden, so steht in dem 32 Bit Wert das Wort **load**.

5.3.3 Applikation

In diesem Abschnitt finden Sie Fragen, die sich mit dem Umgang der CANopen-Bibliotheken aus der Applikationssicht beschäftigen.

F1: Gibt es eine Möglichkeit die PDOs von einem CANopen Slave zu überwachen? Der Slave unterstützt kein Heartbeat und kein Nodeguarding. Ich möchte sehen ob der Slave z.B. alle 50ms ein PDO Telegramm sendet. Wie kann ich das machen?

A: Immer wenn ein RxPDO empfangen wird, wird die Ausgangsvariable **bNewMessage** gesetzt. Ansonsten hat diese Variable keinerlei Funktion.
Geht der Wert der Variablen von 0 auf 1, triggern Sie Ihren Überwachungstimer und setzen die Variable zurück. Da es sich um die Ausgangsvariable eines FBs handelt, müssen Sie zum zurücksetzen der Variablen wie folgt verfahren:

```
pB  : Pointer TO BOOL;

pB  := ADR(pCanOpenPDO_Rx[0].bNewMessage);
pB^ := FALSE;
```

F2: Wie kann man RTR-Frames applikationsgesteuert senden?

A: In den **CanOpen implicit Variables** (Ressourcen->Globalen Variablen) finden Sie eine Liste aller in der Steuerungskonfiguration gemappten CAN RxPDOs (pCanOpenPDO_Rx). Mit Hilfe der Aktion **SendRTRFrame** ist es möglich ein RTR-Frame an das entsprechende PDO zu senden.

```
pCanOpenPDO_Rx[x].bNewMessage := FALSE;
pCanOpenPDO_Rx[x].SendRTRFrame;
If pCanOpenPDO_Rx[x].bNewMessage = FALSE Then
    ...
END_IF
```

5.3.4 Mögliche Fehler, Ursachen und deren Behebung

Hier finden Sie eine Beschreibung von Fehlern, die im Umgang mit dem 3S-CANopen-Stack auftreten können.

F1: Die Ausgänge werden nicht gesetzt. Woran kann das liegen?

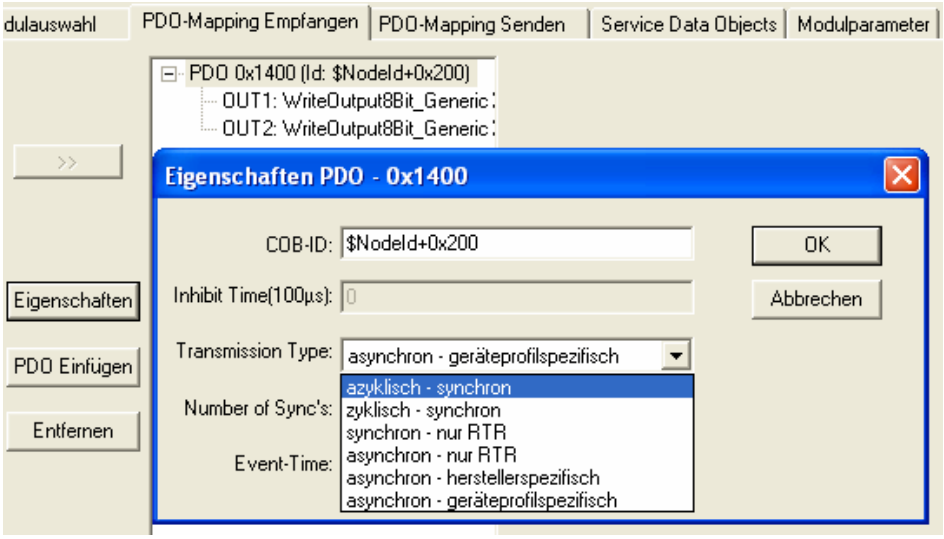
A: Prüfen Sie bitte folgendes:
Verwendung von Variablen
Damit die Ausgangs-Variablen upgedated werden ist es wichtig, dass diese zyklisch in Ihrem Programm aufgerufen werden.
Gehen Sie dazu Bitte zunächst in die Steuerungskonfiguration und geben Sie Ihrem Ausgang einen Namen (z.B.: byOut1).
Rufen Sie diesen Ausgang dann in Ihrem Programm auf.

```
byOut1;
```

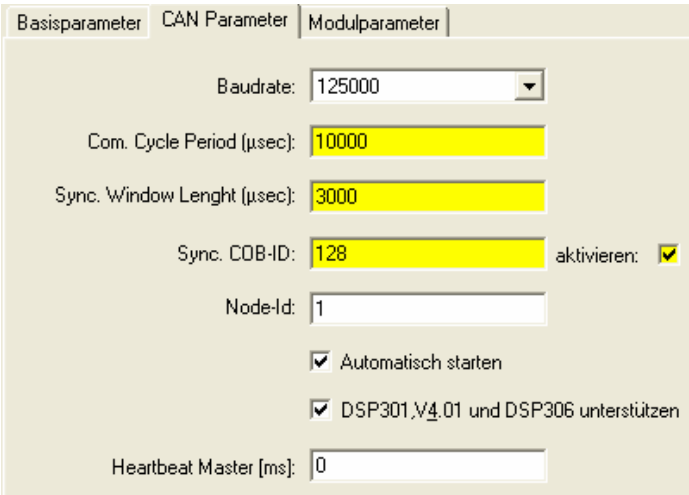
Im Online-Modus können Sie nun den Wert der Variablen ändern und mit [Strg]+[F7] in die

Steuerung schreiben.

Transmission Type
Es kann sein, dass es sich in Ihrem Fall um ein synchrones PDO handelt. Dieses wird nur dann übertragen, wenn gleichzeitig die Sync-Funktionalität aktiviert ist.
Dies können Sie prüfen, wenn Sie sich die Eigenschaften des entsprechenden PDO's ansehen.



Soll das PDO synchron übertragen werden, so muß zusätzlich die Sync-Funktionalität aktiviert sein (gelb markierte Felder).



Tip:
Geben Sie für die **Com. Cycle Period** ein ganzzahliges Vielfaches Ihrer Taskzykluszeit an. Wenn Sie z.B. eine Taskzykluszeit von 10 ms eingestellt haben, dann empfiehlt es sich für die Com. Cycle Period eine Zeit von 10000 µs, 20000 µs, … einzustellen.

Änderungshistorie

Version	Beschreibung	Datum	
1.0	Erstellung	04.11.2005	
1.0	Review: • Einheitliche Schreibweise für: CANopen, Sync(h), Nodeguarding, Messages oder Nachrichten. • Kap. 2.1: Blockweiser, segmentierter SDO-Transfer funktioniert nicht?! • Kap. 2.2.2: Bild fehlt. • Kap. 2.4.1: Kommentar bei „nStatus“ ist abgeschnitten. • Kap. 2.4.4.7.2: END_IF fehlt. • Kap. 4.1.3: Aktueller Dialog, in dem die Felder Prüfsumme übertragen, Bestätigter Transfer, Request beim Bootup und Bootup Requests beantworten disabled sind.	28.11.2005	
1.0	Einarbeiten der Review-Änderungen zusammen mit Fhu	29.11.2005	
1.0	Formale Überarbeitung, Release	01.12.2005	
1.1	Überarbeitung (mit CoDeSys v2.3.6.0): Kap. 3.3 (#5445), Kap. 5.2 neu (#5465), Kap. 2.4.4.2 (#5449, #5450), Kap. 3.6.1(#5446)	11.01.-24.06.2005	
1.1	Formaler Review, Release	24.06.2005	
1.2	Überarbeitung nach Testreport vom 12.06.06 (formale Punkte)	12.06.2006	
1.2	Überarbeitung nach Testreport vom 12.06.06 + #6419	12.07.2006	
1.2	Formaler Review, Übertragung in englische Version, Release	12.07.2006	
1.3	FAQs in Anhang hinzugefügt (#5634), Kap.2.6 korrigiert und erweitert (#6149)	27.09.2006	
1.3	Review O.K.	27.09.2006	
1.3	Release	28.09.2006	
1.4	Chap. 2.2.4 bzgl. “Start All Nodes” (#7092)	27.04.2007	
1.4	Review für #7092	27.04.2007	
1.4	Release	03.05.2007	

© 3S - Smart Software Solutions GmbH