Universität Rostock

Fakultät für Ingenieurwissenschaften
Fachbereich Elektrotechnik & Informationstechnik
Institut für Automatisierungstechnik

Großer Beleg

Entwicklung eines CANopen-Netzwerkes und Funktionsnachweis anhand einer Beispielkonfiguration

Bearbeiter:
cand. ing. Jan Blumenthal

Betreuer:
Prof. Dr.-Ing. Norbert Stoll
Dr.-Ing. Andreas Richter

Inhaltsverzeichnis

OPTOLOGIC
Mess- und Systemtechnik
GmbH
Joachim-Jungius-Str. 9
18059 Rostock

Studienarbeit: Großer Beleg

Thema: **Entwicklung eines CANopen-Netzwerkes für Zeilenkameras und Funktionsnachweis anhand einer Beispielkonfiguration**

Die OPTOLOGIC Mess- und Systemtechnik GmbH entwickelt und fertigt Intelligente Zeilenkameras für die industrielle Prozessautomation. Für komplexe Anordnungen ist es notwendig, mehrere Kameras miteinander zu vernetzen, welches über einen CANopen-Bus erfolgen soll. Der erste Schritt bei der Lösung dieser Aufgabe, wurde durch die Entwicklung einer CAN-API durch Herrn Blumenthal im Rahmen des Kleinen Beleges geschaffen.

Aufgabenstellung:

Aufbauend auf der CAN-API ist das CANopen Protokoll weitestgehend nach CiA 301 Standard umzusetzen. Die Implemetierung ist plattformunabhängig vorzunehmen. Diese Aufgabenstellung beinhaltet sowohl die zeitunkritische Kamerakonfiguration als auch die zeitkritische Übermittlung von Detektions- und Messergebnissen von den Kameras.

Als Basis aller weiteren Arbeitsschritte ist eine Möglichkeit für die Überwachung und die Administration des CANopen-Netzwerkes zu schaffen.

Im ersten Teil der CANopen-Umsetzung soll die Konfiguration der Kameras ermöglicht werden. D.h. es sollen einige repräsentative Parameter in die Kameras übertragen und aus den Kameras gelesen werden können. Die Übertragung dieser Parameter ist nicht echtzeitrelevant. Eine Segmentierung der Zeichenketten bei komplexen Parametern bzw. größeren Datenströmen ist zu ermöglichen.

Die Kommunikation im eigentlichen Prozessbetrieb der Kameras ist zeitkritisch. Besteht die Aufgabenstellung beispielsweise in der Überwachung eines Bahngutes auf Fehler, müssen die vorverarbeiteten Fehlerdaten jeder Kamera zu einem Hostrechner übertragen werden. Bei zeitnahem Auftreten von Fehlern führt das zu einem entsprechendem Zugriff auf den Bus. Eine verzögerungsarme und sichere Übertragung aller Daten ist trotzdem zu gewährleisten. Um das Busverhalten bei dieser Konstellation zu testen und zu optimieren soll ein Beispielnetzwerk aus mehreren Komponenten aufgebaut werden. Kameraseitig sollen Fehler simuliert und daraus Datenströme generiert werden. Das Busverhalten ist zu protokollieren und mögliche Fehler erkannt und eliminiert werden.

Weiterführend soll eine Möglichkeit geschaffen werden, das komplette Netzwerk im nicht-echtzeitfähigen Betrieb über eine Leitrechner fernzuwarten. Dazu ist die Implementierung eines Gateways zur Tunnelung der Daten über TCP/IP erforderlich.

Betreuer OPTOLOGIC:	Dr.-Ing. Andreas Richter	Tel. 0381 / 40 59 490
Beginn der Bearbeitung:	01.06.2001	

Abkürzungsverzeichnis

ACK	Acknowledge (Bestätigung eines Paketes)
ACR	Acceptance code register
AMR	Acceptance mask register
API	Application programable interface, einheitliche Programmierschnittstelle
ARQ	Automatic Repeat Request
C166	16-Bit Mikrocontroller von Infineon
CAN	Controller Area Network
CAN_ID	Datentyp für CAN-Identifier, abhängig vom CAN Modus (11 bzw. 29 Bit Identifier)
CANopen	Protokoll der Anwendungsschicht für den CAN-Bus
CCD	Charge-coupled device, hochauflösender Sensor
COB	Communication Object, bezeichnet eine CAN-Nachricht bei CANopen.
COB-ID	ID einer CAN-Nachricht bei CANopen
CRC	Cyclic Redundancy Checksum
CSDO	Client SDO, Nutzer eines SDO-Services
DeviveNET	Protokoll der Anwendungsschicht für den CAN-Bus
GDI	Graphical Device Interface
ID	Identifier
ISO	International Standard Organisation
Kbs	Kilobit/s
Mbs	Megabit/s
MUX	Multiplexor, dient zur Adressierung eines Eintrages im Objekt Dictionary. Ein MUX besteht aus Index (16Bit) und SubIndex (8Bit)
NMT	Network Management, Dienst von CANopen zur Konfiguration, Initialisierung und Fehlerbehandlung in CANopen-Netzwerken
OSI	Open Systems Interconnection
PC98	Spezifikation von Microsoft und Intel über Aufbau von Standard-PC's
PCI	Peripheral components interconnect
PDO	Process Data Object, Datenobjekt zur Übertragung von Echtzeitdaten bei CANopen.
QT	C++ Klassenbibliothek von Trolltech zur einfachen Erstellung plattformunabhängiger Anwendungen
RPDO	Receive PDO, PDO Anforderung an ein fremdes Gerät
RS232	Serielle Schnittstelle
SDO	Service Data Object, Datenobjekt zur Übertragung von nicht echtzeit-relevanten Daten.
SJA1000	Controller zum Anschluß an CAN-Feldbus
SMP	Symmetric Multi Processing – Betrieb mit mehreren Prozessoren
SoF	Start of Frame
SSDO	Server SDO, Anbieter eines SDO-Services
TCP	Transport Control Protocol
TPDO	Transmit PDO, PDO Übertragung an ein fremdes Gerät
VCI	Virtual programable interface. API für Zugriff auf CAN-Komponenten von IXXAT.
VPN	Virtual Private Network
[x][y]	Adressierung eines Eintrages des Object Dicionaries mit Index=x und SubIndex=y.

Abbildungsverzeichnis

Tabellenverzeichnis

1 Präzisierung der Aufgabenstellung

1.1 Einleitung

Die OPTOLOGIC Mess- und Systemelektronik GmbH entwickelt und fertigt intelligente CCD-Zeilenkameras für die industrielle Prozessautomation. Die CCD-Zeilenkameras beinhalten neben dem Sensor auch einen modular aufgebauten Mikrocomputer mit eigenem Bussystem und einem Mikrocontroller C166 von Infineon. Dieser Computer dient der digitalen Signalverarbeitung der Meßwerte, die größtenteils in Hardwaremodulen vorgenommen wird, um höhere Performancegewinne zu erzielen. Je nach Kundenwunsch können weitere spezielle Module hinzugeordert werden, die die Funktionalität der Kamera auf das entsprechende Einsatzgebiet optimieren.
In der Vergangenheit wurden die Ergebnisse der Meßwertanalyse über eine serielle Verbindung an einen PC übertragen. Da diese Lösung für komplexe Aufgaben mit mehreren Überwachungsanforderungen äußerst unflexibel ist, besteht nun die dringende Aufgabe, ein Netzwerkmodul nach Industriestandard zu fertigen. Für dieses Einsatzgebiet sind Feldbussysteme prädestiniert. Ein Erweiterungsmodul mit einer CAN-Bus Funktionalität wurde entwickelt.

Die Wahl der Kommunikationshardware fiel auf den CAN-Bus, da er in weiten Teilen der Industrie eingesetzt wird, z.B. dem Automobilbau. Dadurch wird die Integration der Kameras in bestehende Systeme deutlich erleichtert.
Die wesentliche Eigenschaft des CAN-Busses ist die Echtzeitfähigkeit, die für die Anwendungen der CCD-Zeilenkameras von entscheidender Bedeutung sind, um Ergebnisse und vor allem Fehlermeldungen in einem definierten Zeitfenster erfolgreich zu übertragen. Das CAN-Protokoll weist allerdings auch eine Vielzahl von Schwächen auf, die erst in komplexeren Applikationen deutlich werden.
Die Programmierung der Treibersoftware für die CAN-Module erfolgte in einer Studienarbeit des studentischen Mitarbeiters Jan Blumenthal [1]. Hierbei wurde Wert auf Plattformunabhängigkeit und eine einheitliche API (Application programable interface) gelegt.

Der CAN-Bus ist nach dem ISO-OSI Schichtenmodell ein Layer 2 Protokoll (Data Link Layer). Es beinhaltet keinerlei Mechanismen zur Übertragung von größeren Datenströmen, zur Fehlerkontrolle, Ausfallsicherheit und Synchronisation von Netzwerkteilnehmern. Diese Aufgaben werden laut ISO Spezifikation von höheren Schichten übernommen. Angepaßte Protokolle für den CAN-Bus waren lange Zeit nicht standardisiert. Die Folge war eine Vielzahl proprietärer, firmeneigener Protokolle bzw. kompletter Eigenentwicklungen, die die Einbindung von CAN-Applikationen in bestehende, fremde Netzwerke deutlich erschwerten.
Derzeit existieren für den CAN-Bus zwei speziell entwickelte und standardisierte Protokolle, die die genannten Anforderungen erfüllen, *DeviceNET* und *CANopen*. Im Rahmen mehrerer Projektsitzungen wurde entschieden, CANopen zu verwenden, da es sich um eine Entwicklung handelt, die sich zunehmend als europaweiter Standard etabliert, wogegen DeviceNET vorwiegend in den USA vertreten ist.

Bei der Umsetzung des CANopen-Protokolls in Software wurden zwei Alternativen diskutiert. Einerseits gab es die Möglichkeit, fertige Bibliotheken für zehntausende von D-Mark zu erwerben. Diese Libraries sind allerdings nicht für jede zu unterstützende Plattform vorhanden. Gegen die Bibliotheken sprachen außerdem die Lizenzmodelle, die teilweise spätere Lizenzgebühren pro Gerät an Kunden von OPTOLOGIC vorsahen bzw. die fehlende Möglichkeit der Einflussnahme auf den Objektcode, z.B. bei späteren Erweiterungen. Die zweite Möglichkeit war die Eigenentwicklung einer dem Standard entsprechenden Software. Aus Kostengründen wurde Variante zwei favorisiert.

Die Aufgabe dieser Studienarbeit besteht darin, eine Software für die verschiedenen Hardwareplattformen zu entwickeln, die sich weitestgehend an den CANopen Standard anlehnt und bei der Umsetzung die firmeneigenen Interessen berücksichtigt.

1.2 Die Einsatzszenarien

Die Einsatzgebiete der Kameras in der Industrie sind vor allem kontinuierliche Überwachungsvorgänge, z. B. die Erkennung von Oberflächenstörungen. Jede Kamera kann bei komplexen Aufgaben aufgrund ihrer Fähigkeiten nur einen gewissen Bereich beobachten. Ist dieser Bereich kleiner als das zu kontrollierende Gebiet, müssen mehrere Kameras versetzt die Überwachung übernehmen, wodurch Überlappungseffekte auftreten, die nur durch eine nachgeschaltete digitale Signalverarbeitung verarbeitet werden können. Die Kameras nehmen eine Vorauswertung der Aufnahmen vor und übertragen die Ergebnisse in der Regel zu einem Leitrechner, meistens ein handelsüblicher PC mit CAN-Karte (s. Abbildung 1-1). Denkbar ist auch eine Masterkamera, die eine Endauswertung vornimmt und die Ergebnisse weiterleitet (s. Abbildung 1-2).

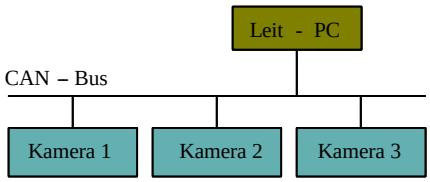

Abbildung 1-1 Leit-PC steuert 3 Netzwerkteilnehmer (Standardanwendnung)

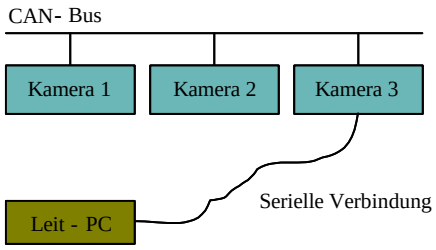

Abbildung 1-2 Steuerung des Netzwerks über eine serielle Verbindung

1.3 Die Hardwareplattformen

Die Kamera von OPTOLOGIC arbeitet, wie eingangs erwähnt, mit einem C166 Mikrocontroller, der bist zu 16MB addressieren kann. Allerdings ist in den Systemen aus Kostengründen üblicherweise deutlich weniger bestückt. Da die CPU auch für die Echtzeitverarbeitung der Daten herangezogen wird, darf die CANopen Implementierung nur relativ wenig Resourcen (Rechenzeit, Speicherverbrauch) in Anspruch nehmen.
In Abbildung 1-1 ist ein PC gezeigt, der eine CAN-Karte enthält. Die Steuerung des Netzwerkes erfordert die Entwicklung einer CANopen-Software für diesen PC. Eine Betrachtung hinsichtlich des Speicherverbauchs ist bei solchen PC's nicht notwendig, da RAM-Bausteine in Megabyte-Bereichen deutlich weniger kosten als für Embedded PC's. Die Performancebetrachtung ist vernachlässigbar, da ein aktueller Mikroprozessor um Größenordnungen schneller als ein C166 ist. Um einen Echtzeitbetrieb zu ermöglichen, ist zu beachten, dass auch ein passendes Betriebssystem gewählt wird, welches u.U. auf virtuellen RAM als unkalkulierbare Performancebremse verzichtet.
Die Implementierung des CAN-Protokolls gestaltet sich in einem PC anders als in Mikrocontrollern, da jeder Hersteller einer CAN-Karte seinen eigenen Betriebssystemtreiber mitliefert, auf den in der Applikation bzw. der CANopen-Softwarer zugegriffen werden muss. Dieser Zugriff ist betriebssystemabhängig und herstellerabhängig. Um der CANopen-API eine einheitliche Schnittstelle zu allen CAN-Controllern in PC's und Mikrocontrollern zur Verfügung zu stellen, wurde in einer vorbereitenden Studienarbeit eine CAN-API entwickelt, die den kompletten Zugriff auf die CAN-Controller kapselt[1].

1.4 Vorüberlegungen zur Software

Die Verwendung von Netzwerken in der industriellen Praxis zeigt eine Fülle weiterer Möglichkeiten zur effizienten Nutzung von Resourcen auf.
In Abbildung 1-3 ist ein CAN-Netzwerk inkl. Master-Kamera dargestellt, das per Leit-PC gesteuert wird. Angenommen, in dem Netzwerk tritt während des Bootens ein Fehler auf, kann es äußerst kostspielig sein, vor Ort nach dem Fehler zu suchen, vor allem, wenn dazu erst größere Anfahrtswege durch geschultes Personal nötig sind. Viel preisgünstiger ist es, sich *remote* über ein ‚Virtual Private Network' (VPN) in die fehlerhafte Anlage einzuwählen und zu versuchen, den Fehler zu beheben. Um dies zu ermöglichen, sind allerdings weitere Gedanken zur Konzeption der Software nötig. Neben der reinen CANopen Implementierung wird die Entwicklung zusätzlicher Softwaremodule für die Kameras und eines Administrationsprogramms nötig. Dieses Programm muss so konzipiert werden, dass es sich *remote* steuern lässt.

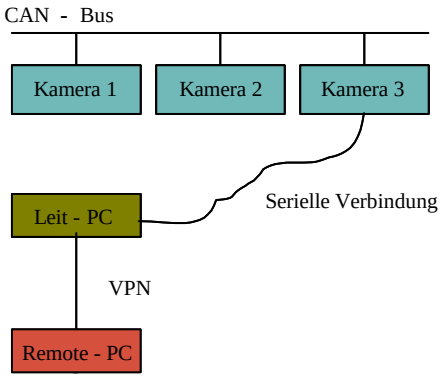

Abbildung 1-3 Fernwartung eines Netzwerkes

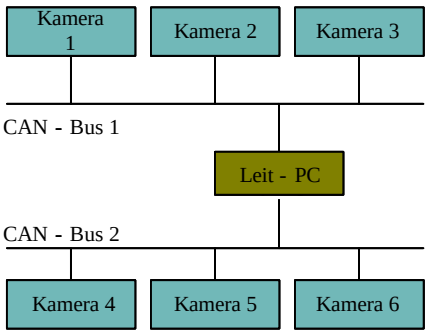

Abbildung 1-4 Administrierung mehrerer CANopen Netzwerke

In Abbildung 1-4 ist ein Leit-PC dargestellt, der gleichzeitig zwei CAN-Netzwerke administriert. Das kann sinnvoll in Zusammenhang mit der Überwachung von parallelen Prozessen sein, z.B. mehrerer Förderbänder. Für den Systembetreuer ergeben sie dadurch verschiedene Einsparungspotenziale, die in Tabelle dargestellt sind.

Ein PC	Mehrere PCs
Einmaliges Installieren des Betriebssystems	Mehrmaliges Installieren des Betriebssystems
Wartung eines PC's	Wartung mehrerer PC's
Bedienung eines PC's	Bedienung mehrerer PC's
Administrierung durch eine Anwendung (Multiple documents interfaces[MDI] in MFC Anwendungen)	Administrierung durch mehrere Anwendungen (Single Document Interfaces[SDI] in MFC-Anwendungen
Hoher Hardwareaufwand	Niedriger Hardwareaufwand pro PC

Tabelle 1 Einsparungspotenziale durch Verwendung eines Leit-PC's

Dies erfordert ein dynamisches Verhalten der CANopen-API, da die Applikation auf dem PC mehrere CANopen-Instanzen unterstützen muss. Die API muss in der Lage sein, jede Instanz völlig unabhängig von der anderen zu konfigurieren und zu behandeln.

2 Theorie von CANopen

2.1 Der CAN-Bus

Der Feldbus CAN, abgeleitet aus der Bezeichnung Controller Area Network, steht für ein serielles Bussytem. Er wurde 1987 von der Robert Bosch AG in erster Linie für die Vernetzung von Komponenten im Automobilbau entwickelt. Der CAN-Bus zeichnet sich vor allem durch folgende Eigenschaften aus:

Eigenschaft	Beschreibung
Broadcastfähigkeit	Nachrichten werden grundsätzlich als Broadcast an alle Teilnehmer verschickt. Die Teilnehmer haben die Möglichkeit, durch Filter im CAN-Controller das Auslösen von Interrupts bei Eintreffen von unerwünschten Nachrichten zu unterbinden.
Multimasterfähigkeit	Jede angeschlossene Station ist in der Lage, die Buskontrolle zu übernehmen und eine Übertragung zu starten.
Echtzeitfähigkeit	Durch eine bitweise Busarbitrierung nach CSMA-CA wird die Echtzeitfähigkeit erreicht. Die Nachricht mit dem höchsten Identifier setzt sich bei gleichzeitigem Beginn zweier Übertragungen am Bus durch [1].
Nachrichten-identifizierung	Nachrichten werden nicht von einem Sender an einen Empfänger geschickt, sondern mit Identifiern per Broadcast an alle Teilnehmer gesendet. Über den Identifier wird gleichzeitig die Priorität festgelegt.
Zuverlässigkeit	Das Versenden von kurzen Frames (8 Bytes) verringert das Risiko von Fehlübertragungen.
Standardisierung	Die Standardisierung nach ISO DIS 11519-1 und ISO DIS 11898 ermöglicht den problemlosen Einsatz in CAN-Netzwerken anderer Hersteller.
Fehlererkennung	Der Einsatz von CRC-Fehlercodes und Bit-Stuffing ermöglicht eine hohe Fehlererkennung.
Preis	Geringe Kosten durch Einsatz von Standardkomponenten.

Tabelle 2 CAN-Bus Eigenschaften

Der Standard des CAN-Bus definiert lediglich den Frameaufbau, das Arbitierungsverfahren sowie einfache Fehlererkennungmechanismen. Das Übertragungsmedium ist im CAN-Protokoll nicht spezifiziert. Als Bus wird überlicherweise eine verdrillte Zwei-Drahtleitung verwendet, auf die die Teilnehmer parallel aufgesetzt werden. Es ist aber auch möglich, optische Übertragungsmedien zu benutzen.
Die Kameras von OPTOLOGIC verwenden aus Kostengründen eine Zweidrahtleitung. Dazu ist es erforderlich, in den angeschlossenen Geräten die Tranceiver optisch von den CAN-Controllern zu trennen, um Potenzialunterschiede auszugleichen.
Die Übertragung von Frames erfolgt in differenziellem Manchestercode. Der CAN-Bus erlaubt in Abhängigkeit von der Leitungslänge programmierbare Bitraten von bis zu 1 MBit/s (40m). Bei einer Buslänge von 620m sind noch 100 KBit/s möglich. Eine theoretische Begrenzung der Teilnehmerzahl gibt es nicht. Allerdings ist durch den Einsatz von Standardkomponenten nach RS485 eine Obergrenze von 32 Teilnehmern zu empfehlen.

2.2 Einleitung

Die aufgezeigten Eigenschaften des CAN-Bus charakterisieren ihn als Protokoll der Bitübertragungs- und der Sicherungsschicht. Es fehlen Mechanismen zur Übertragung längerer Datenblöcke, zur Überwachung von Stationen, zur Synchronisation von Daten und Stationen sowie ein effizientes Netzwerkmanagement, d.h jede Anwendung die CAN in komplexeren Situationen einsetzt, muss eigene proprietäre Mechanismen anwenden, um diese Nachteile zu beheben, wodurch sie inkompatibel zu allen anderen Produkten auf dem Markt wird.
Die CiA (Can in Automation) hat den Standard ***CANopen*** definiert, um diese Dienste auf eine einheitliche standardisierte Grundlage zu stellen. CANopen ist ein Schicht 7 Protokoll, d.h. es beinhaltet Aufgaben der Anwendungsschicht. Eine grobe Übersicht über das Referenz-Modell zeigt Abbildung 2-1. Die linke Seite der Darstellung zeigt das ISO-OSI Schichtenmodell. Auf der rechten Seite sind die Funktionsmodule zu erkennen, die in CANopen definiert sind.

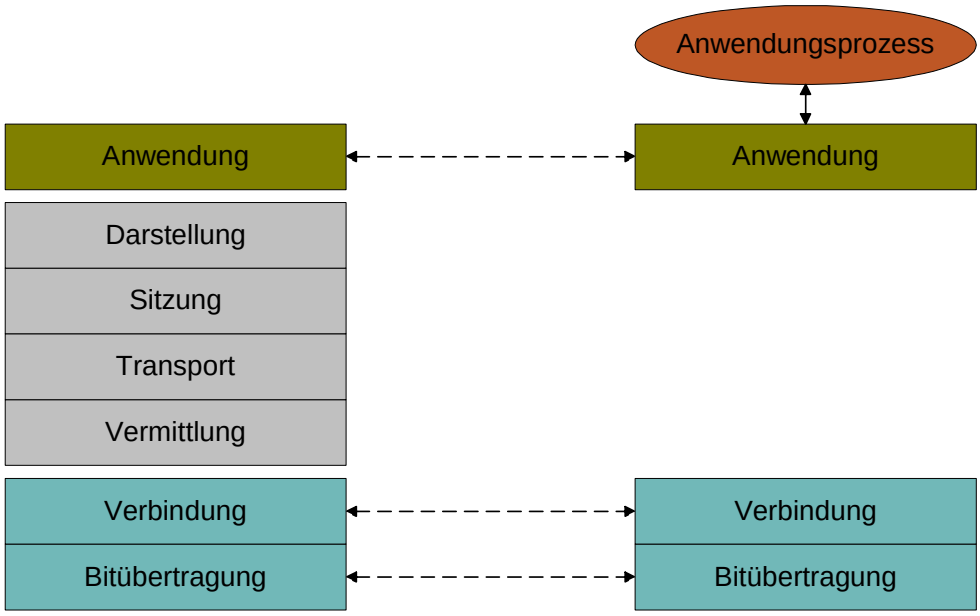

Abbildung 2-1 Referenzmodell von CANopen[1]

Die Schichten 3 bis 6 sind in CANopen nicht definiert, da eine Vielzahl von Aufgaben, wie z.B. Auf- und Abbau von Sitzungen (definiert in der Sitzungsschicht), nicht benötigt werden oder in CANopen eindeutig festgelegt sind. So ist die Kommunikationsdarstellungsschicht nicht vorhanden, da im Standard geregelt ist, wie Daten gespeichert und übertragen werden.[2] Die Dienste der Vermittlungsschicht werden ebenfalls nicht benötigt, da ein CAN-Netzwerk nicht routbar ist, d.h. es ist immer ein abgeschlossenes System. Daten müssen nicht über die Netzwerkgrenzen hinaus übertragen werden. Einige Aufgaben der vermeintlich nicht definierten Schichten sind allerdings in der Anwendungsschicht umgesetzt, wie z.B.

- Segmentieren von Daten
- Fehlerkontrolle
- Netzwerkmanagement

[1] Die Bitübertragungs- und die Verbindungsschicht sind Inhalt der CAN-Spezifikation [11]
[2] CANopen benutzt zur Speicherung und Übertragung von Daten den von Intel bekannten ‚little endian style'

In CANopen ist die Schnittstelle der Anwendungsschicht zur Verbindungsschicht nicht definiert, d.h die Umsetzung ist implementierungsabhängig. Die Kommunikation zwischen Prozessanwendung und Anwendungsschicht erfolgt über Dienste und Objekte. Die genaue Umsetzung der API ist jedoch nicht spezifiziert. Bei der SUN Workstation Karte[15] wird z.B. die komplette Anbindung über das typische Unix-Treibermodell realisiert[3]. In anderen Implemetierungen werden Anwendungschicht und Anwendung statisch verbunden[16].

Der Datenaustausch zwischen Anwendung und Anwendungsschicht erfolgt über 4 *Dienste:*

- ***Request*** Eine Anfrage, die von der Anwendung ausgelöst wird, um mit der CANopen-Schicht zu kommunizieren.
- ***Indication*** Ein Anzeichen, das von der Anwendungsschicht ausgelöst wird, um der Anwendung ein internes Ereignis, z.B. die Ankunft von neuen Daten zu melden.
- ***Response*** Repräsentiert eine Antwort, die von der Anwendung initiiert wird, um eine Indication zu beantworten.
- ***Confirmation*** Eine Bestätigung, die von der Anwendungsschicht ausgelöst wird, um den Erhalt einer Response zu melden.

Innerhalb von CANopen wird zwischen vier verschiedenen Servicetypen unterschieden:

- ***Local Service*** arbeitet lediglich mit der lokalen CANopen –Schicht zusammen, ohne Datenübertragung.
- ***Unconfirmed Service*** bezeichnet eine Übertragung an andere Netzwerkteilnehmer, die nicht bestätigt wird. Es sind auch Broadcasts möglich.
- ***Confirmed Service*** ist ein Dienst, um Daten zu einem Netzteilnehmer zu übertragen. Das Ergebnis dieser Operation wird an die Anwendung zurückgegeben (Confirmation).
- ***Provider Initiated Service*** wird von der Anwendungsschicht ausgelöst (Indication), z.B. bei Eintreffen von Nachrichten, die nicht vom lokalen Gerät initiiert worden sind.

Die Dienste *Confirmed Service* und *Unconfirmed Services* bezeichnet man allgemein als *remote services*, da es sich um Dienste handelt, die über das Netzwerk abgewickelt werden.

2.3 Das Gerätemodell

Ein CANopen-Gerät ist in drei große Funktionsmodule unterteilt (Abbildung 2-2):

- ***Anwendung*** Die Anwendung umfasst die komplette Funktionalität des Gerätes. Sie ist die Schnittstelle zur Prozessumgebung.
- ***Kommunikation*** Das Kommunikationsmodul stellt alle Funktionen zur Übertragung von Daten über das CAN-Netzwerk zur Verfügung. Die Anwendung hat keine Möglichkeit, direkt auf das Netzwerk zuzugreifen.
- ***Object Dictionary*** Das Object Dictionary ist eine geräteabhängige, interne Datenbank. Sie ist das Bindeglied zwischen Kommunikationseinheit und Anwendung.

[3] Zugriffe auf das User-Interface der CANopen Karte von Stock Microsysteme werden über die Standard Device Treiber Library Aufrufe open(), close(), pread(), pwrite() und ioctl() realisiert.

Die komplette Beschreibung des Object Dictionaries (Einträge, Datentypen, Wertebereiche) und der Anwendung (Aufgabe, Definition der Prozessumgebung) bezeichnet man als ***Geräteprofil (device profile)***.

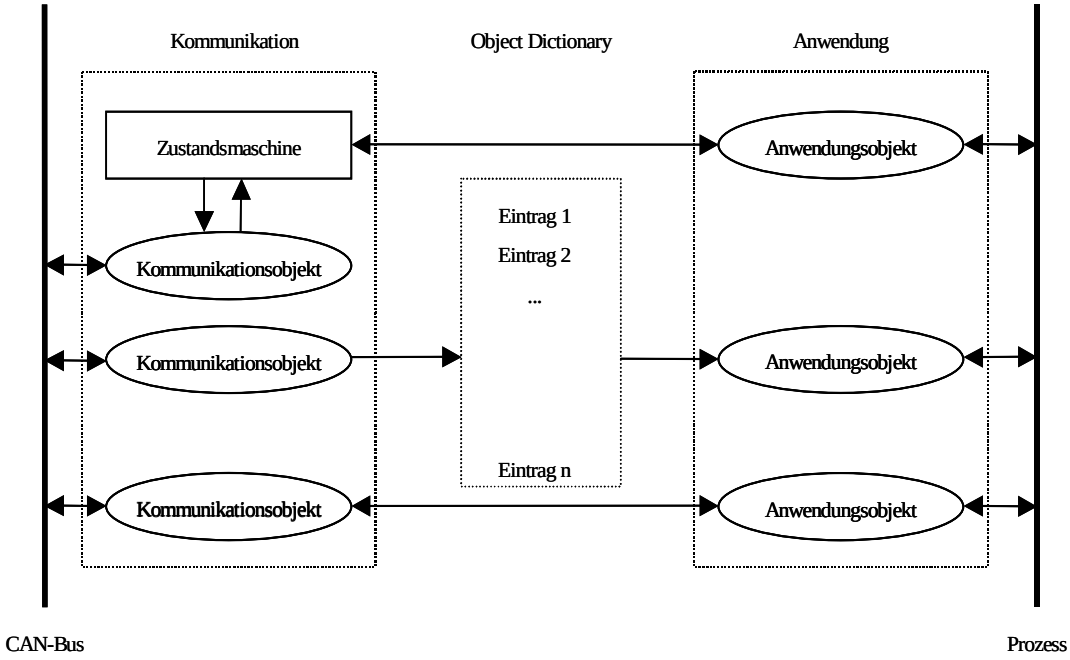

Abbildung 2-2 CANopen Gerätemodell

2.4 Das Object Dictionary

Der wichtigste Teil eines Geräteprofils ist das Object Dictionary. Es handelt sich dabei um eine relationale Datenbank. Das Datum der Datenbankeinträge beinhaltet vordefinierte oder selbstdefinierte Datentypen, wie Integer, Arrays und Strukturen. Die Einträge werden mittels eines 16-Bit Indizies addressiert, sodass sich maximal 65536 Objekte ergeben. Zu beachten ist, dass weite Bereiche der Indizes vordefiniert sind (s. Abbildung 2-3) und demzufolge nur für jeweiligen Zweck benutzt werden dürfen. Die anwendungsspezifischen Indizes liegen im Bereich von 0x2000 bis 0x5fff. Ein Zugriff auf das Object Dictionary eines fremden Gerätes über das Netzwerk ist möglich.
Die Adressierung von Elementen in komplexeren Strukturen, z.B. Records oder Arrays erfolgt über SubIndizies. Diese liegen im Bereich von 0-255 und sind immer vom Typ BYTE. Die Kurzschreibweise für Zugriffe auf das Object Dictionary ist ***Eintrag[Index][SubIndex]***. Das Parameterpaar [Index][SubIndex] wird auch als ***Multiplexor (MUX)*** bezeichnet. Im Falle der Adressierung über einen Index ohne Subindex wird bei einfachen Datentypen, z.B. int, sofort auf die Variable zugegriffen. Bei einer Struktur wird dagegen die komplette Datenstruktur angesprochen. Dazu ist es erforderlich, die Einträge einer Struktur im RAM direkt hintereinander anzuordnen (siehe Kap. 4.9 S. 35). Die Subindizies für einfache Datentypen, wie Integer8, müssen immer Null sein. Die Umsetzung des Object Dictionaries ist im Standard nicht definiert und daher implementierungsabhängig.
Durch die willkürliche Festlegung der Indizies ergeben sich deutliche Nachteile bei der Adressierung der Einträge, die immer in leichten Bursts auftreten. Im durchnummerierten Fall könnte auf ein Element wie in einem Array zugegriffen werden. Das entspricht einer Multiplikation des

Indizies mit der Datentypbreite des Arrayelements und anschließender Addition des Arrayzeigers. Bei CANopen muß dagegen ein aufwändiger Suchalgorithmus implementiert werden. Das kostet Prozessorzeit – vor allem bei Mikrocontrollern. In der Praxis führt dies zu unsauberer Programmierung. [4]

Index	Objekt
0	nicht benutzt
0x0001-0x001F	Statische Datentypen
0x0020-0x003F	Komplexe Datentypen
0x0040-0x005F	Hersteller spezifische komplexe Datentypen
0x0060-0x007F	Geräte Profil spezifische statische Datentypen
0x0080-0x009F	Geräte Profil spezifische komplexe Datentypen
0x00A0-0x0FFF	Reserviert
0x1000-0x1FFF	Kommunikations Profil
0x2000-0x5FFF	Hersteller spezisches Profil
0x6000-0x9FFF	Geräteprofil standardisierter Geräte
0xA000-0xFFFF	Reserviert

Abbildung 2-3 Struktur des Object Dictionaries

2.5 Das Kommunikationsmodell

Im Kommunikationsmodell sind die verschiedenen Kommunikationsobjekte und die verschiedenen Modi der Nachrichtenübertragung definiert. CANopen unterstützt das Senden von synchronen und asynchronen Nachrichten. Synchrone Nachrichten dürfen nur zu definierten Zeiten verschickt und beantwortet werden. Zur Übertragung von synchronen Nachrichten sind zwei zusätzliche Objekte definiert worden (SYNC-Object, Time Stamp Object). Das Senden von asynchronen Nachrichten ist jederzeit möglich.

CANopen unterscheidet drei Typen von Kommunikationsbeziehungen:

- ***Master/ Slave*** Innerhalb des Netzwerkes gibt es ein Gerät, das einen Dienst bereitstellt und auf alle anderen Geräte (*Slaves*) per Request zugreift. Ein Request von seiten der Slaves ist nicht möglich. Beispiel: Network Management
- ***Client/ Server*** Diese Beziehung gilt zwischen zwei Geräten, die einen Datenaustausch durchführen. Der Client löst einen Request an den Server aus (z.B. Download/Upload). Nach Beendigung der Aufgabe sendet der Server eine Antwort. Die Beziehung ist bidirektional, d.h. Client und Server können auch in entgegengesetzter Richtung kommunizieren, vorausgesetzt, beide unterstützen die Dienste entsprechend. Beispiel: Service Data Object (SDO).

[4] Der Zugriff auf das Object Dictionary kann z.B durch ein simples Switch-Case Konstrukt erfolgen[16]. Dies stellt mit Sicherheit die Triviallösung dar und ist nicht im Sinne einer modularen, zukunftorientierten Softwareentwicklung.

- ***Producer/ Consumer*** Bei dieser Beziehung kommuniziert ein Producer mit keinem oder mehreren Consumern. Dabei wird zwischen dem ***Push model*** und dem ***Pull model*** unterschieden. Bei dem Push model wird die Kommunikation vom Producer initiiert und von den Consumern empfangen, aber nicht beantwortet. Bei dem Pull model hingegen wird die Kommunikation von den Consumern initiiert. Der Producer empfängt die Nachricht und antwortet mit einer ***Response***.

2.6 Datentypen

In Abbildung 2-1 ist das Modell von CANopen dargestellt. Das Fehlen der Darstellungsschicht, dem Programmteil, der die Daten in das entsprechende Format der Hardware umwandelt (little endian style bzw. big endian style) setzt voraus, dass Sender und Empfänger Kenntnis von der Struktur der Datentypen haben müssen, um Daten auszutauschen.
Alle Datentypen sind definiert als Sequenz von aufeinanderfolgenden Bits. Das Versenden von Daten erfolgt, wie erwähnt, im ‚little endian style' (Abbildung 2-4), d.h. niederwertigste Byte 1 wird als erstes übertragen, gefolgt von dem Byte 2 bis zu Byte k.

Byte	1	2	K
Inhalt	Bit_7 - Bit_0	Bit_{15} - Bit_8	Bit_{8*k-1} - Bit_{8*k-8}

Abbildung 2-4 Darstellung einer Bitsequenz im 'little endian style'

Grundsätzlich gibt es bei CANopen vier verschiedene Datentypen:

- ***Basic Data Types*** In diese Kategorie fallen einfache Datentypen wie Boolean, Integer8 (Wertebereich 2^8), Integer16, Integer24, Unsigned8, Real32
- ***Compound Data Types*** Bezeichnet eine Sammlung gleicher Basistypen, allg. bekannt als Array, oder verschiedener Basistypen, bekannt als Struktur bzw. Record.
- ***Extended Data Types*** Die erweiterten Datentypen sind abgeleitet aus Basis- und zusammengesetzten Datentypen. Beispiele sind:
  - **Visible_String**, bezeichnet ein Array aus darstellbaren Buchstaben vom Typ Visible_Char, die im Bereich von 0x20-0x7F liegen (8-Bit).
  - **Unicode_String**, bezeichnet ein Array aus darstellbaren Buchstaben vom Typ Unsigned16 (16Bit) zur Unterstützung asiatischer Schriftzeichen.
  - **Time_of_Day**, Record, das die absolute Zeit seit dem 01.01.1984 lokaler Zeit repräsentiert. Dieses Record besteht aus folgenden Basistypen:
    Struct of
    Unsigned28 ms,
    void4 reserved,
    unsigned16 days,
    Time_of_Day
- ***Domains*** Ein Datentyp, der von CANopen nicht genauer spezifiziert ist. Die Behandlung innerhalb der CANopen-Schicht ist völlig implementierungsabhängig. Dieser Datentyp ist besonders sinnvoll im Einsatz von großen veränderlichen Datenmengen, die sehr applikationsspezifisch sind (siehe S. 34 – Einführung von entfernten Objekten).

2.7 Kommunikationsobjekte

In CANopen sind eine Reihe von speziellen Objekten und Diensten definiert, die die Applikation bei den unterschiedlichsten Aufgaben, wie Echtzeitnachrichten oder Konfigurationsnachrichten, unterstützen.

2.7.1 PDO – Das Process Data Object

Das ***Process Data Object (PDO)*** wurde entwickelt, um die Echtzeitfähigkeit von CANopen zu gewährleisten. Die Übertragung der 512 möglichen PDO-Nachrichten erfolgt ohne Protokolloverhead. Der Inhalt der max. 8 Datenbits einer CAN-Nachricht kann aus mehreren Einträgen des Object Dictionaries bestehen, die bitweise hintereinander gesendet werden. Die einzelnen Objekte einer PDO-Nachricht werden in ***Mapping Parametern*** spezifiziert – einem Element im Object Dictionary.
In Tabelle 3 ist ein Beispiel für ein einfaches PDO-Mapping gezeigt. Der Eintrag mit Sub-Index 0 beinhaltet die Anzahl der Elemente, die dieses Record besitzt. In diesem Fall wird die PDO-Nachricht aus drei Objekten zusammengesetzt. In den weiteren Einträgen (1-3) der Struktur wird jeweils in einem long-Wert die Information gespeichert, wie der Index/SubIndex des jeweiligen Objektes kodiert ist und mit welcher Größe das Objekt in die PDO-Nachricht gemappt wird. In dem gezeigten Beispiel werden demnach die Einträge [0x2000][0], [0x2001][1] mit einer Länge von 8 Bit=1 Byte bzw. 16 Bit=2 Byte sowie Eintrag [0x2002][0] mit einer Länge von 32Bit=4 Byte der Reihe nach übertragen. Das Mapping der einzelnen Objekte erfolgt ebenfalls im ‚little endian style'.[5] Die Länge der Daten in dieser Nachricht beträgt 7 Byte.
Die Einträge in der Mapping-Struktur sind üblicherweise konstant (***Statisches PDO-Mapping***). Es besteht zusätzlich die Möglichkeit, diese Struktur dynamisch dem Prozess anzupassen (***dynamisches PDO-Mapping***. Das setzt allerdings voraus, dass sich alle Producer und Consumer dynamisch anpassen können, denn die Empfänger müssen in der Lage sein, die übertragenen Daten wieder auseinander zu nehmen. Das Ändern der Mapping Parameter kann nur im Status NMT_PreOperational durchgeführt werden (s. Kap. 2.8 S.17).

Sub-Index	Datentyp	Inhalt			Bedeutung
		Index	Sub-Index	Länge	
0	BYTE	3			3 Object Dictionary Einträge werden in die PDO-Nachricht gemappt
1	ULONG32	0x2000	0x00	0x08	Objekt [0x2000][0] wird mit einer Länge von 8 Bit=1 Byte in die PDO-Nachricht gemappt
2	ULONG32	0x2001	0x01	0x10	Objekt [0x2001][1] wird mit einer Länge von 0x10=16 Bit=2 Byte nach dem ersten Eintrag in die PDO-Nachricht gemappt
3	ULONG32	0x2002	0x00	0x20	Objekt [0x2002][0] wird mit einer Länge von 0x20=32 Bit=4 Byte nach dem zweiten Eintrag in die PDO-Nachricht gemappt

Tabelle 3 Beispiel für PDO-Mapping Parameter

[5] Demzufolge sehen die Datenbytes der Nachricht im gezeigten Beispiel wie folgt aus: $b1_7$-$b1_{0,}$ $b2_7$-$b2_0$, $b2_{15}$-$b2_8$, $b3_7$-$b3_0$, $b3_{15}$-$b3_8$, $b3_{23}$-$b3_{16}$, $b3_{31}$-$b3_{24}$.

Zu jeder PDO Nachricht existiert neben den Mapping Parametern noch eine Struktur mit ***Communication Parametern***, in der Parameter zur eigentlichen Übertragung festgelegt sind.

Sub-Index	Kategorie	Beschreibung
0	Verbindlich	Anzahl der unterstützten Subindizies
1	Verbindlich	COBID der PDO-Nachricht
2	Verbindlich	Übertragungstyp
3	Optional	Timeout
4	Optional	Reserviert
5	Optional	Event timer

Tabelle 4 Struktur der PDO Communication Parameters

Bei PDO-Nachrichten wird zwischen fünf verschiedenen Übertragungstypen unterschieden, wobei Kombinationen einzelner Typen teilweise möglich sind:

- Synchrone Nachrichten — Nachrichten, die periodisch und synchron zu einem SYNC-Objekt verschickt werden
- Asynchrone Nachrichten — PDO wird nicht synchron zum SYNC-Objekt übertragen
- RTR — Remote transmission requests werden nur auf Anfrage von anderen Geräten verschickt
- Zyklisch(n) — Nachrichten werden synchron, aber nur nach Auftreten von n SYNC-Objekten, verschickt.
- Azyklisch — Nachrichten werden synchron zum SYNC-Object verschickt, allerdings nur, wenn intern ein Event aufgetreten ist.

Die Indizies der PDO Mapping und Communication Parameters in dem Object Dictionary werden nach Tabelle 5 berechnet. ***RPDO*** steht dabei für ***Receive PDO*** und bedeutet, dass dieses Gerät die entsprechende PDO-Nachricht empfangen kann. ***TPDO*** steht für ***Transmit PDO***, d.h. das angegebene PDO kann übertragen werden.

Name	Index
RPDO communication parameter index	0x1400 + pdonumber –1
RPDO mapping parameter index	0x1600 + pdonumber –1
TPDO communication parameter index	0x1800 + pdonumber –1
TPDO mapping parameter index	0x1a00 + pdonumber –1

Tabelle 5 Berechnung der PDO Parameter

Das Versenden von PDO-Messages kann auf drei verschiedene Arten ausgelöst werden:

- Events durch die Applikation
- Ablauf eines Timers (s. Tabelle 4)
- Eintreffen einer asynchronen Anfrage von einem anderen Gerät (RTR)

Die vordefinierte Schnittstelle der CANopen API bietet der Applikation zwei Funktionen, ***ReadPDO()*** und ***WritePDO()***, zur Interaktion mit PDO's an. Die Write-Funktion dient dem unquittierten Senden von Nachrichten (***Push model***). Als Argument erwartet die Funktion lediglich die PDO-Nummer und die bereits gemappten Daten. Es gibt keinen Rückgabewert. Der Service ReadPDO erwartet als Argument ebenfalls die PDO-Nummer. Hier sendet der Consumer einen Request an den PDO-Producer. Dieser quittiert die Nachricht mit den erforderlichen Daten (max. eine CAN-Nachricht). Als Ergebnis kehrt der Service mit den gesendeten Daten zurück (***Pull model***).
Um echte Performance zu erhalten, sollte dem WritePDO Service der Vorrang gegeben werden, da er die Applikation nicht durch unnötiges Warten blockiert.

2.7.2 SDO – Das Service Data Object

Durch das Service Data Object (SDO) ist ein bidirektionaler Zugriff auf das Object Dictionary eines jeden Gerätes über das Netzwerk möglich. Die Übertragung ist eine klassische Client/Server Anwendung. Alle Übertragungsanforderungen werden vom Client gestartet. Als Server ist in diesem Fall der Besitzer des Object Dictionaries festgelegt. Grundsätzlich erfolgt die Übertragung von Daten in mehreren ***Segmenten***. Bevor allerdings Daten ausgetauscht werden, erfolgt der Aufbau einer Verbindung. Ist diese erfolgreich abgeschlossen, werden die Segmente nach dem Hand-Shake Verfahren übertragen, d.h. Daten werden zum Client geschickt und danach mit einer Antwort quittiert. Die Windowsize beträgt 1, nach einem Datenpaket wird ein Antwortpaket verschickt. Durch das Handshake-Verfahren ergibt sich automatisch die Notwendigkeit, auf Antwortnachrichten zu warten (***Timeout***). Die Umsetzung von Timeouts ist implementierungsabhängig und in CANopen nicht weiter spezifiziert. Allein aus dieser Tatsache ergibt sich, dass SDO's nicht in echtzeitkritischen Programmteilen benutzt werden sollten. Für SDO's, deren Datenlänge max. 4 Bytes beträgt, existiert die Möglichkeit, die Daten beschleunigt zu übertragen (***expedited***). In diesem Fall erfolgt die Übertragung bereits in der Initialisierungsphase, ein Handshake ist trotzdem notwendig. Im normalen Betrieb können pro Nachricht bis zu 7 Nutzbyte gesendet werden. In dem 8. Byte sind Steuerinformationen enthalten, z.B. ein Toggle-Bit, das bei jeder neuen Nachricht vom Client negiert wird - eine einfache und aber nicht sehr sichere Möglichkeit, Fehler in der Übertragung zu entdecken. Tritt ein Fehler auf, wird die Übertragung per ***AbortSDO*** abgebrochen. AbortSDO ist ein Service, um aufgebaute Verbindungen jederzeit zu trennen. Er kann vom Client und vom Server gleichmassen benutzt werden. Die AbortSDO-Nachricht enthält vordefinierte 32-Bit Fehlercodes, die allerdings nicht sehr aussagekräftig sind und durch die fehlende Durchnummerierung nur mittels Switch/Case Konstrukten ausgewertet werden können. Am Ende einer Übertragung haben Client und Server zusätzlich die Möglichkeit, über eine optionale CRC-Checksumme zu prüfen, ob der Transfer erfolgreich abgeschlossen wurde.
Optional bietet CANopen die Möglichkeit, 127 Segmente in Blöcke zusammenzufassen. Nach dem Senden von 127 Segmenten, wird über ein zusätzliches Hand-Shake die Nummer des letzten empfangenen Segments und eine CRC-Checksumme des Blocks in einer speziellen Nachricht übermittelt. Bei fehlerhaften Übertragungen kommt der Go-Back-n ARQ (Automatic Repeat Request) zur Anwendung, d.h. die Übertragung wird nach dem letzten fehlerfreien Blocktransfer wiederholt. Es ist nicht notwendig, die kompletten Daten neu schicken.

Die SDO-Services benutzen die in Tabelle 6 gezeigten Attribute, die je nach Bedarf den einzelnen Services als Argument übergeben werden. Die ***SDO-Nummer*** entspricht dabei der Node-ID des Servers. Der Multiplexor dient der Adressierung des Elements bzw. SubElementes im Object Dictionary. Der ***Transfertype*** legt fest, ob beim Nachrichtenaustausch auch Angaben über die Größe der Daten übertragen werden. Die Einstellung ***Normal*** bewirkt eine Angabe der Länge und verlängert damit die Übertragung, wogegen ***Expedited*** auf einen Größenangabe verzich-

tet. Bei üblichen Datentypen wie char, short, long kann so der Datenaustausch auf ein einfaches Handshake verkürzt werden. Bei größeren Datenmengen ist immer ein mehrfacher Handshake notwendig.

Attribut	Beschreibung
SDO Number	SDO-Nummer, entspricht der NodeID (0...127)
User type	Client oder Server
Multiplexor	Struktur aus einem Index (16 Bit) und SubIndex (8 Bit), die den Eintrag im Object Dictionary adressiert
Transfer type	abhängig von der Datenlänge: - *Expedited* (Beschleunigt), Datenlänge max. 4 Bytes - *Normal*, bei einer Länge über 4 Bytes

Tabelle 6 Attribute der SDO Services

Innerhalb von CANopen werden vier SDO-Services unterschieden, die intern in mehrere Unterservices aufgeteilt sind:

- SDO Download
  Durch diesen Service werden Daten in Segmente zu einem Server übertragen. Dieser Service wird bestätigt und gibt als Ergebnis eine Fehlermeldung zurück. Abhängig von der Datenlänge wird eine *beschleunigte* oder *normale* Übertragung vorgenommen. Dieser Service muß implementiert sein und wird unterteilt in:
  - Initiate SDO Download
  - Download SDO Segment
- SDO Upload
  Dieser Service dient zum Übertragen von Daten in Segmenten vom Server zum Client. Er wird ebenfalls mit Fehlermeldung bestätigt und liefert als Ergebnis zusätzlich die Daten zurück. Dieser Service muß implementiert sein. Er wird unterteilt in:
  - Initiate SDO Upload
  - Upload SDO Segment
- SDO Block Download
  Durch diesen Service werden Daten in Blöcken, die jeweils 127 Segmente enthalten, zu einem Server übertragen. Dieser Service wird bestätigt und liefert als Ergebnis eine Fehlermeldung. Dieser Service ist optional und wird unterteilt in:
  - Initiate SDO Block Download
  - Download SDO Block
  - End SDO Block Download
- SDO Block Upload
  Dieser Service dient zum Übertragen von Daten vom Server zum Client in Blöcken zu je 127 Segmenten. Er wird mit einer Fehlermeldung und den übertragenenen Daten quittiert. Dieser Service ist ebenfalls optional und muß nicht implementiert sein. Er wird unterteilt in:
  - Initiate SDO Block Upload
  - Upload SDO Block
  - End SDO Block Upload

2.7.3 Das SYNC-Object

In echtzeit-relevanten Netzwerken kann es wichtig sein, eine gemeinsame Zeitbasis zu besitzen und Nachrichten synchron bzw. zu einem festgelegten Zeitpunkt zu verschicken. Um dies zu ermöglichen, wird periodisch ein SYNC-Object von einem SYNC-Master verschickt. Dies kann ein beliebiges Gerät innerhalb des Netzwerkes sein und ist nicht an andere Master-Funktionalitäten gebunden. Beim SYNC-Object spricht man wie bei PDO's von einer ***Producer/ Consumer*** Beziehung. Die Sendung eine SYNC-Nachricht entspricht einem unbestätigtem Broadcast. Die Consumer haben nach Erhalt eines SYNC-Objects die Möglichkeit, PDO-Nachrichten *synchron* zu verschicken.
Um zu verhindern, dass andere CAN-Nachrichten beim der Arbitrierung der SYNC-Nachricht den Bus übernehmen, besitzt dieses Objekt eine CAN-ID mit einer sehr hohen Priorität. Trotzdem ist zu beachten, dass es durch laufende Übertragungen zu einer kurzen Latenzzeit kommen kann, durch die die Übertragung möglicherweise leicht verzögert werden kann.
Die Datenlänge eines SYNC-Objekts ist null, sodass lediglich der Protokollokopf einer CAN-Nachricht verschickt wird.

2.7.4 Das Time Stamp Object

Geräte in einem Netzwerk sind in der Regel getaktet. Da aufgrund von Fertigungstoleranzen die Periodendauer eines Quarzes unterschiedlich ist und über längere Zeit zu deutlichen Zeitunterschieden zwischen den einzelnen Geräten führen kann, ist es erforderlich, in bestimmten Abständen eine Synchronisation der Teilnehmer des Netzwerkes durchzuführen.
Das Time Stamp Object dient zur Übertragung der aktuellen Uhrzeit. Die zu übertragende Nachricht enthält die 6 Bytes große Struktur TIME_OF_DAY und entspricht einem unbestätigtem Broadcast. Das Time Object entspricht ebenfalls der ***Producer/ Consumer*** Beziehung. Eine Zeitzone ist nicht festgelegt. Es gilt lokale Zeit.

2.7.5 Das Emergency Object

Das ***Emergency Object*** ist ein Notfall Objekt. Bei internen Fehlern wird vom Producer eine Notfallnachricht mit einem entsprechenden Fehlercode verschickt. Mögliche Ursachen können z.B. sein:

- Reset
- Spannungsabfall
- CAN-Bus Überlauf
- Fehlerhafte oder verloren gegangene Pakete
- Überschreiten von Temperaturgrenzen

Zu beachten ist, dass SDO-Objekte ein eigenes Notfallsystem beinhalten und durch das Emergency Object nicht abgedeckt werden. Notfallobjekte werden nur einmal an ein oder mehrere Consumer per Broadcast verschickt und nicht bestätigt. Die Behandlung der Fehler in den Consumern ist in CANopen nicht weiter spezifiziert, sie erfolgt in der Anwendung. Die Implementierung ist optional.

2.8 Netzwerk Management

Das ***Netzwerk Management (NMT)*** ist ein Überwachungs- und Monitoringservice innerhalb von CANopen. In jedem Netzwerk gibt es einen ***NMT-Master*** und max. 126 ***NMT-Slaves***. Jedes Gerät eines CANopen Netzwerkes besitzt eine NodeID (1...127) und beinhaltet eine NMT-Statemachine, die sich in vier verschiedenen Zuständen befinden kann (s. Tabelle 7).

NMT Zustand	Beschreibung
NMT_INITIALISING	Konfiguration von Geräte- und Kommunikationsparametern, z.B. der Standard-Übertragungsrate. Netzwerkaktivität ist nicht erlaubt.
NMT_PRE-OPERATIONAL	Synchronisation des Netzwerkes, z.B. Einstellung der Systemzeit und Setzen einer systemweit einheitlichen Übertragungsrate. Initialisieren von netzwerkabhängigen Einträgen im Object Dictionary.
NMT_OPERATIONAL	Gerät ist vollständig in Netzwerkaktivität eingebunden und funktionstüchtig. Alle Netzwerkservices können in Anspruch genommen werden.
NMT_STOPPED	Gerät ist vom Netzwerk abgekoppelt und reagiert lediglich auf NMT Anweisungen.

Tabelle 7 Zustandsmodi eines NMT-Gerätes

Beim Booten eines NMT-Gerätes befindet es sich im Status NMT_INITIALISING. Das Gerät führt unabhängige Initialisierungen durch. Danach wechselt es automatisch in den Zustand NMT_PRE-OPERATIONAL und sendet einen ***Bootup-Event***, eine CAN-Nachricht mit dem Identifier NMT_Error_Control+NodeID (s. Tabelle 10). Dadurch ist gewährleistet, dass jedes Gerät im Netzwerk Kenntnis von einer Neuinitialisierung dieses Gerätes hat. Im Modus NMT_PRE-OPERATIONAL sind Netzwerkaktivitäten erlaubt, die zur Konfiguration und Synchronisation des Netzwerkes erforderlich sind. Bei einer erfolgreichen Initialisierung des gesamten Netzwerkes kann der NMT-Master das Netzwerk in den Modus NMT_OPERATIONAL schalten. In diesem Modus startet die eigentliche Prozessaufgabe des Netzwerkes. Aus diesem Grund sind nur in diesem Modus PDO's erlaubt.
Für die Steuerung der NMT-Geräte stehen 4 ***Network Control Services*** zur Verfügung (s. Tabelle 8). Als Argument wird dabei immer die entsprechende NodeID übergeben. Eine NodeID von null entspricht einem Broadcast.

Service	Beschreibung
Start Remote Node	Setzt den NMT-Status auf OPERATIONAL. Dieser Service kann nicht lokal ausgeführt werden.
Stop Remote Node	Stoppt das Gerät. Es wird vom Bus abgekoppelt und geht in Status NMT_STOPPED über.
Enter Pre-Operational	Gerät wechselt in Status NMT_PRE-OPERATIONAL
Reset Node	Startet das Gerät neu und geht automatisch in Status NMT_INITIALISING über.
Reset Communication	Startet die Kommunikationumgebung neu und geht automatisch in den Zustand NMT_INITIALISING über.

Tabelle 8 NMT Services

Zur Überwachung der Zustände von NMT-Slaves wurden in CANopen 2 ***Error Control Services*** definiert. Sie basieren beide auf der periodischen Übertragung von Nachrichten über den CAN-Bus. Diese Nachrichten besitzen den Identifier NMT_Error_Control+NodeID. Die Implementierung eines Dienstes ist verbindlich.

- ***Life guarding*** Der NMT-Master sendet periodisch Nachrichten an den Slave, der diese in einer festgelegten Zeit (life guarding time) beantworten muß (***Pull model***). Antwortet der Slave nicht, wird die Applikation des NMT-Master darüber informiert. Sollte der NMT-Slave keine Nachricht vom NMT-Master erhalten, wird seine Applikation über das Ausbleiben der Nachricht informiert.

- ***Heartbeat*** Ein Gerät im Netzwerk sendet periodisch Nachrichten aus, die nicht bestätigt werden (***Push model***). Alle anderen Geräte des Netzwerkes sind Consumer und müssen bei einem Ausbleiben des Herzschlages ihre Anwendungen informieren.

2.9 Netzwerkinitialisierung

Das Booten des gesamten Netzwerkes bedarf gesonderter Betrachtung. Dieser Vorgang wird durch den NMT-Master kontrolliert. Eine Übersicht über den prinzipiellen Ablauf zeigt Abbildung 2-5.

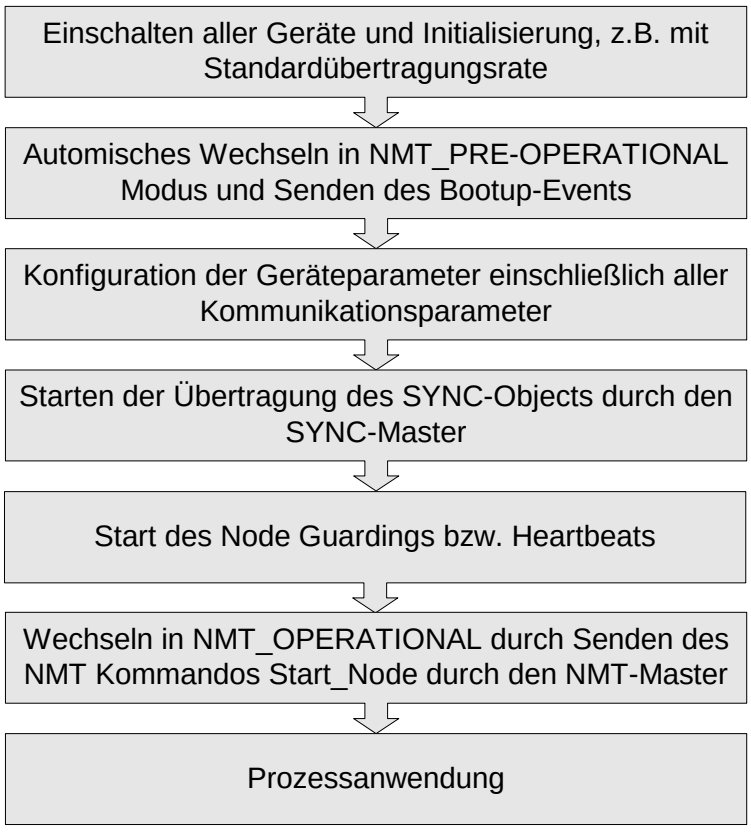

Abbildung 2-5 Initialisierung eines CANopen Netzwerkes

2.10 Das Pre-Defined Connection Set

Um den Konfigurationsaufwand für einfache Netzwerke zu vereinfachen wurde das ***Pre-defined Connection Set*** definiert. In diesem Set sind die Standard-Identifier für die CAN-Nachrichten der CANopen Objekte festgelegt. Die Identifier sind direkt nach dem Eintritt in den Modus NMT_PRE-OPERATIONAL verfügbar und gelten sowohl in Netzwerken mit 11-Bit Identifiern als auch in Netzwerken mit 29-Bit Identifiern.

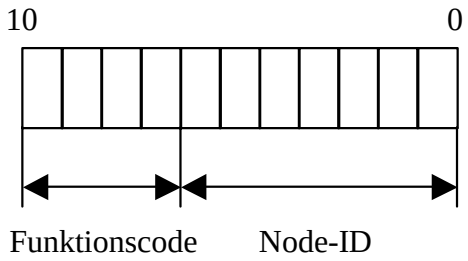

Abbildung 2-6 Definition des COB-Identifiers im pre-defined connection set

Die CAN-Identifier sind in einen Funktionsteil und einen Node-ID Teil aufgeteilt, der das Gerät adressiert (s. Abbildung 2-6). Eine Node-ID von null entspricht einem Broadcast.

Objekt	Funktionscode (binär)	COB-ID
NMT	0000	0
SYNC	0001	80h
TIME STAMP	0010	100h

Tabelle 9 Broadcast Objekte im pre-defined connection set

Objekt	Funktionscode (binär)	COB-ID
EMERGENCY	0001	081h – 0FFh
PDO1 (tx)	0011	181h – 1FFh
PDO1 (rx)	0100	201h – 27Fh
PDO2 (tx)	0101	281h – 7FFh
PDO2 (rx)	0110	301h – 37Fh
PDO3 (tx)	0111	381h – 3FFh
PDO3 (rx)	1000	401h – 47Fh
PDO4 (tx)	1001	481h – 4FFh
PDO4 (rx)	1010	501h – 57Fh
SDO (tx)	1011	581h – 5FFh
SDO (rx)	1100	601h – 67Fh
NMT Error Control	1110	701h – 77Fh

Tabelle 10 Peer-to-Peer Objekte im pre-defined connection set

Durch die Definition der Node-ID in dem Identifier wird die Möglichkeit geschaffen, peer-to-peer Verbindungen aufzubauen.
Der Funktionsteil (Funktioncode) legt die Objekt-Priorität festlegt. Die Arbitrierung von CAN-Nachrichten erfolgt über den CAN-Identifier, der bitweise vom MSB zum LSB übertragen wird. ***Low*** ist der dominante Zustand auf dem CAN-Bus, d.h. Nachrichten, die im MSB eine Null haben, setzen sich eher auf dem Bus durch als Nachrichten mit einer Eins. Durch geeignete Zuordnung von Identifiern wird demnach die Echtzeitfähigkeit verbessert. Aus Tabelle 10 geht hervor, dass PDO's eine höhere Priorität haben als SDO's. Im umgekehrten Fall würden SDO's die Übertragung von PDO's behindern und dadurch die Anwendung gefährden.

Aus Tabelle 10 geht weiterhin hervor, dass ein Gerät im ***pre-defined connection set*** 4 PDO's(tx) und 4 PDO's(rx) unterstützt. Somit ergeben sich die in Kap. 2.7.1 erwähnten 512 verschiedenen PDO-Nachrichten, wenn berücksichtigt wird, dass der 128. als Broadcastteilnehmer benutzt wird.
PDO(tx) sind PDO-Nachrichten, die nach dem ***Push model*** übertragen werden, d.h. sie werden nicht beantwortet. Es können pro Gerät max. 4*8 Byte in PDO(tx) übertragen werden. PDO(rx) werden für eine PDO-Anforderung benutzt (***Pull model***).
Die Nummerierung in Tabelle 10 ergibt, dass PDO1(tx) von Gerät 1 höher priorisiert ist als PDO1(tx) von Gerät 2, welches aber wiederum höherpriorisert ist als PDO2(tx) von Gerät 1. Durch diese Verschachtelung ist der Entwickler in der Lage, Nachrichten der verschiedenen Geräteanwendungen nach Gerät und nach Wichtigkeit zu sortieren.

3 Die CAN-API

3.1 Einführung

Die in der Aufgabenstellung geforderte Plattformunabhängigkeit der CANopen Software erfordert eine möglichst einfache Anpassung der Software an verschiedene CAN-Controller. Die Voraussetzungen wurden durch die Definition einer einheitlichen CAN-API, die in einer Studienarbeit [1] erarbeitet wurden, geschaffen. Durch Nutzung dieser Funktionen entfällt eine Anpassung der CANopen Software an verschiedene CAN-Controller, da die Kapselung der Hardware vollständig in der CAN-API durchgeführt wird.
Eine Übersicht der zur Verfügung stehenden Softwaremodule ist in Abbildung 1-1 dargestellt. Die Kapselung in Module (CAN, CANopen) und die Definition eindeutiger Schnittstellen zwischen ihnen ermöglicht den Austausch einzelner Module ohne Änderung der anderen Komponenten. Die einmal entwickelte Applikation läuft demnach ohne Probleme mit einem anderen CAN-Controller zusammen. Es müssen nur die beiden unteren Treiber-Schichten ausgetauscht werden. Alle direkten Hardwarezugriffe auf den CAN-Controller befinden sich im CAN-Treibermodul. Ein direkter Zugriff auf den CAN-Bus von der CANopen-Schicht oder sogar von der Applikationsschicht aus ist nicht gestattet. Daraus ergibt sich der Vorteil, dass die CANopen-Schicht komplett hardwareunabhängig ist und der größte Teil der darauf aufsetzenden Applikationen ebenfalls.
Die CAN-Schicht ist ohne weiteres auch für eine '*normale*' CAN-Umgebung nutzbar. Sie ist unabhängig von der CANopen-Schicht.

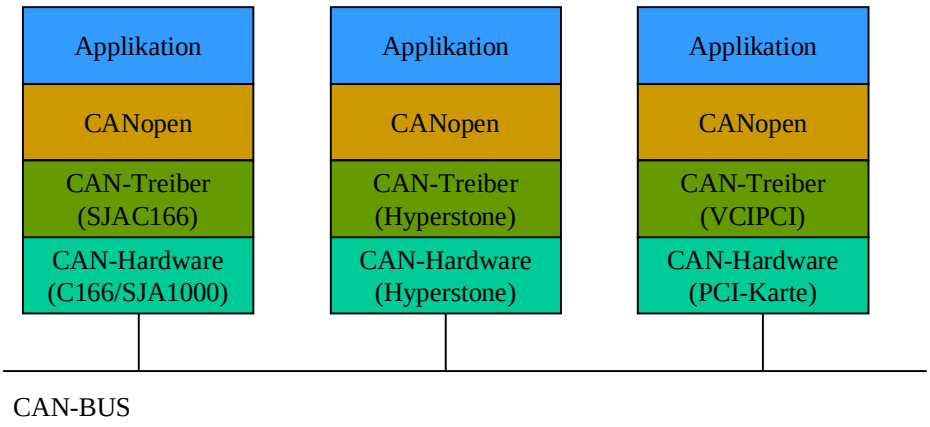

Abbildung 3-1 Komponenten einer CANopen-Anwendung

3.2 Das CAN-Handle

Die CAN-API bildet im Prinzip zu den höheren Schichten einen virtuellen CAN-Controller ab. Die Adressierung und Kommunikation mit der eigentlichen CAN-Hardware erfolgt in den Funktionen der CAN-API, die zur Identifizierung der Hardware ein *CAN_HANDLE* benutzen. Diese Struktur beinhaltet Informationen über die zu benutzende Hardware und den aktuellen Zustand des virtuellen CAN-Controllers (s. Abbildung 3-2). Die Handles sind erforderlich, um mehrere Instanzen der CAN-Schicht voneinander zu trennen. Dadurch ist es möglich, in Appli-

kationen mehrere CAN-Controller gleichzeitig zu anzusprechen. Die Unterscheidung, wie welcher CAN-Baustein angesprochen wird, fällt in der CAN-Treiberschicht.

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// CAN_HANDLE declaration, read only by application layers
typedef struct
{
        CAN_MESSAGE     receivequeue[MAXMESSAGES];
        BYTE            readindex;
        BYTE            writeindex;
        UINT8           flags;                  // e.g. Message-Overrun

        void*           userdata;               // can be used by application layers
                                                // Attention: CanOpen uses this ptr to store the CANOPEN_HANDLE
        //unsigned char  can_mode;              // 0=11Bit Identifier (FUTURE use)
                                                // 1=29Bit Identifier (FUTURE use)
} CAN_HANDLE;
```

Abbildung 3-2 Deklaration des CAN_HANDLE (Basishandle)

Das CAN_HANDLE wurde in zwei Bereiche gegliedert, ein *Basishandle* und ein *abgeleitetes Handle*. Das Basishandle dient in erster Linie zum Speichern von Informationen, die jeden CAN-Controller betreffen, z.B. Ablegen von Nachrichten in Nachrichtenwarteschlangen. Die hardwarenahen Informationen werden in einem abgeleiteten HANDLE gespeichert, auf das nur die CAN-API Zugriff hat. Auf diese Weise wird verhindert, dass andere Funktionsmodule versuchen, direkt auf die Hardware zuzugreifen.
Für die Programmierung von Applikationen der höheren Schichten ist die Kenntnis von Attributen des CAN_HANDLE's, mit der Ausnahme von *userdata*[6], unwichtig. Entscheidend in diesem Zusammenhang ist nur, dass zur Identifizierung des richtigen CAN-Controllers lediglich ein Zeiger auf das Basishandle an die Funktionen der CAN-API übergeben wird.

3.3 Funktionen der CAN-API

Die CAN-API stellt der CANopen Schicht eine Reihe von Funktionen zur Verfügung, um auf die eigentliche Hardware indirekt zugreifen zu können (siehe Abbildung 3-3).

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// CAN-API public function declarations
OPTOERROR       CAN_InitHardware(CAN_HANDLE **handle, CAN_Configcallback configcallback );
OPTOERROR       CAN_FreeHardware(CAN_HANDLE **handle);
OPTOERROR       CAN_InitCan(CAN_HANDLE *handle);
OPTOERROR       CAN_TransmitData(CAN_HANDLE *handle, CAN_MESSAGE *message);
OPTOERROR       CAN_SetReceiveInterrupt( CAN_HANDLE *handle, CAN_Receivecallback callback);
OPTOERROR       CAN_WaitMsgTimeout(CAN_HANDLE *handle, long milliseconds);
OPTOERROR       CAN_SetBaudrate(CAN_HANDLE *handle, int number);
OPTOERROR       CAN_SetFilterMask(CAN_HANDLE *handle, CAN_ID ac_code, CAN_ID ac_mask);
CAN_MESSAGE*    CAN_GetMessage(CAN_HANDLE *handle);
OPTOERROR       CAN_GetStatus(CAN_HANDLE *handle, long *status);
```

Abbildung 3-3 Funktionen der CAN-API

[6] Die Verwendung von *userdata* ist vor allem zur Speicherung von Informationen, z.B. Zeiger auf Strukturen übergeordneten Instanzen sinnvoll.

3.4 CAN-Environment

Das CANEnvironment ist ein Modul, in dem die Hardwaretreiber ausgewählt und die konkreten Hardwareeinstellungen vorgenommen werden. Eine mögliche Einstellung kann z.B. das Setzen der Nummer der Interruptleitung bei externen CAN-Controllern sein. Diese Angabe kann für denselben CAN-Treiber bei den diversen Geräten unterschiedlich sein. Ein Vergleich zum PCI-Bus zeigt, dass alleine durch Umstecken von Einsteckkarten die Interruptleitung wechseln kann. Die Auswahl der CAN-Controller erfolgt durch Übergabe einer Callback-Funktion beim Aufruf von CAN_InitHardware(). Die Callback-Funktionen werden üblicherweise in dem Modul *CANEnvironment.c* definiert. Die Kenntnis der Funktion entspricht prinzipiell dem Namen einer Resource, wie z.B. COM1 bzw. LPT1, die man unter Windows öffnet. In der Callback-Funktion werden die grundlegenden Einstellungen, die für die Initialisierung der CAN-Hardware erforderlich sind, vorgenommen. Es handelt sich im Prinzip um Administratoreinstellungen, wie sie in einer Windows-Systemsteuerung bzw. einer üblichen Treiberinstallation vorgenommen werden.

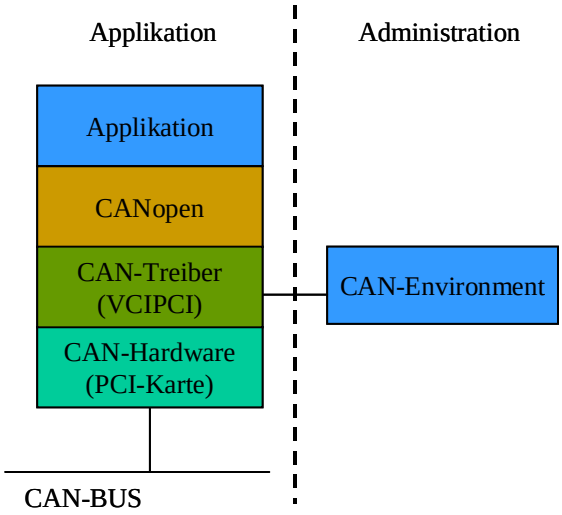

Abbildung 3-4 Trennung von Applikation und Administration

Die Applikation sollte von diesen Einstellungen unberührt bleiben, da sie mit anderen CAN-Controllern ohne Änderungen lauffähig sein soll (s. Abbildung 3-4). Mit der hier vorgestellten Lösung ist bei einem Wechsel des CAN-Controllers lediglich das Modul CANEnvironment auszutauschen.

3.5 Die CAN-Message

Da es sich bei der CAN-Treiberschicht um einen virtuellen CAN-Controller handelt, ist ebenfalls eine virtuelle CAN-Message definiert. Die Struktur der CAN_Message (s. Abbildung 3-5) beinhaltet alle wichtigen Informationen, die nötig sind, um eine Nachricht über den Bus zu senden bzw. zu empfangen, z.B. den Identifier der Nachricht und zu sendenden bzw. zu empfangenen Datenbytes. Zusätzlich werden noch Informationen über den Status der Nachricht abgelegt, z.B. ob es sich um einen *Remoteframe* oder einen *Datenframe* handelt.

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// CAN_MESSAGE declaration (for all CAN-Controllers)
typedef struct
{
        CAN_ID          id;                     // Identifier 11-/29-Bit
                                                // 11 bit in low word/byte, 29 bit complete(future use)
        UINT8           data[MAXDATABYTES];     // Array for up to 8 data(UINT8) bytes
        UINT8           data_length:4;          // number of data bytes (0-8)
        UINT8           remoteframe:1;          // RTR-Bit: 0=Dataframe, 1=Remoteframe
        UINT8           reserved:3;             // not used
        //Future use
        //UINT32        time_stamp;             // Timestamp for receive queue objects
} CAN_MESSAGE;
```

Abbildung 3-5 Deklaration der CAN_MESSAGE

Diese Message-Struktur wird von der aufsetzenden Schicht (CANopen) benutzt, um Nachrichten zu verschicken bzw. per CAN_GetMessage() aus der ReceiveQueue der entsprechenden CAN Instanz auszulesen und zu interpretieren.

4 Die CANopen Software

4.1 Einleitung

Die CANopen Software wurde wie auch die CAN-API in ANSI-C verfasst. Die einzelnen Aufgaben bei der Entwicklung waren:

- ***Funktionsfähigkeit***
  Die Hauptaufgabe besteht in der zuverlässigen Funktion der gesamten CANopen-API, was umfangreiche Tests unter verschiedenen Testbedingungen und Umgebungen erfordert.
- ***Resourcenverbrauch***
  Das Haupteinsatzgebiet der CANopen-API ist der Bereich der eingebetteten Systeme, die üblicherweise mit Resourcen wie CPU-Kapazität und Speicherausbau knapp ausgerüstet sind. Es ist demnach wichtig, die Software möglichst speicher- und performancesparend zu programmieren.
- ***Portabilität***
  Ein weiteres Ziel war, die CANopen-Software so zu verfassen, dass sie unabhängig von Prozessor und Betriebssystem ist. Durch die Trennung der Softwaremodule CAN-API und CANopen-API war es möglich, die CANopen Software komplett unabhängig vom gewählten CAN-Controller zu entwickeln. Alle betriebssystemspezifischen Aufgaben wurden ausgelagert. Sie ist somit vollständig portabel.
- ***Zulassung mehrerer Instanzen***
  Die Unterstützung mehrerer CANopen-Instanzen ermöglicht es, mehrere CANopen-Netzwerke (physikalisch getrennte CAN-Bus Systeme) gleichzeitig zu administrieren. Dies kann für Überwachungsprogramme von parallel laufenden Fertigungsstraßen sehr sinnvoll sein.
- ***Fernwartung***
  Die Möglichkeit der Fernwartung von CANopen Netzwerken eröffnet ungeahnte Kosteneinsparungspotenziale durch den Wegfall von z.T. erheblichen Anfahrts- und Auslösekosten.
- ***Erweiterbarkeit***
  Der Einsatz der CANopen Software in den unterschiedlichsten Anwendungen erfordert ein hohes Mass an Flexibilität und Erweiterbarkeit der Software.

Bei der Erstellung der Software wurden folgende Programmierparadigmen verfolgt:

- ***Vermeidung dynamischer Speicherallozierung***
  Die Aufgabe war, die Software u.a. auch für Mikrocontroller zu entwickeln. Da die entsprechenden Schaltungen meist nur knapp mit RAM ausgestattet sind, erhöht sich im Laufe der Zeit die Gefahr, durch Speicherfragmentierung keinen RAM mehr zugewiesen zu bekommen. Dieser Entwicklung wird vorgebeugt, indem die CANopen API intern komplett ohne dynamische Speicherallozierung auskommt.
- ***Klare API und Typdefinitionen***
  Die API wurde so entwickelt, dass die Funktionen zur Anwendung möglichst aussagekräftig und eindeutig sind. Sie sollen die Anwendung entlasten und nicht verkomplizieren.
  Wichtig war auch die Definition von eindeutigen Typen, wodurch die Möglichkeit einer späteren Erweiterbarkeit gegeben wurde (Bsp: CAN-ID, MULTIPLEXOR, NODEID)

- ***Dokumentation***
  Die Software wurde durchgängig in Englisch dokumentiert, um internationalen Kunden bzw. ausländischen Mitarbeitern ein besseres Verständnis des Quellcodes zu ermöglichen.
- ***Scripting***
  In den Quellverzeichnissen befinden sich unterschiedliche Skripte, die z.B. automatische Clean-Ups durchführen, d.h. Löschen aller durch Kompilierung erzeugten temporären Dateien. Besonders sinnvoll können Skripte auch im Zusammenhang mit größeren CANopen Netzwerken sein, wenn der Entwickler an gemeinsamen Programmcode Änderungen vornimmt. Durch Skripte erhält er die Möglichkeit, alle Projekte auf einmal zu übersetzen und auf die Mikrocontroller herunterzuladen.
- ***Compiler-Switches***
  Das Verhalten der CANopen-API kann durch Compilerschalter (Compilerswitches) in Hinblick auf Geschwindigkeit, Programmlänge und Funktionsumfang enorm beeinflußt werden (s. Kap. 4.12.5).
- ***ANSI-C vs. C++***
  Die CANopen-API wurde komplett in ANSI-C entwickelt, aber nur aus der Tatsache heraus, weil im Mikrocontrollerbereich C++-Compiler selten sind. Trotzdem wurde versucht, einige C++Techniken zumindest ansatzweise anzuwenden.

4.2 Komponenten von CANopen

In Abbildung 4-1 ist der grundsätzliche Aufbau der CANopen Software dargestellt. Die betriebssystem- bzw. hardwarespezifischen Module wurde wie bei der CAN-API ausgelagert, um die Plattformunabhängigkeit der Anwendung und der CANopen-API nicht zu gefährden.

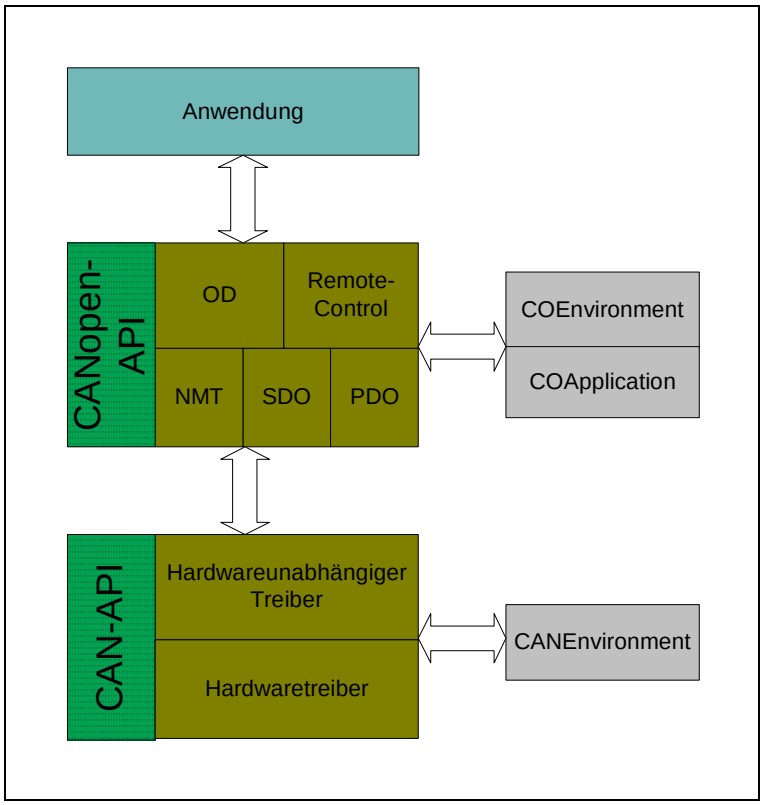

Abbildung 4-1 Komponenten von CANopen

Die Prozessanwendung hat nur Zugriff auf die fünf Teilmodule der CANopen-API und kommun kommuniziert grundsätzlich über fest definierte Schnittstellen. Es gibt keine Möglichkeit, direkt auf die CAN-API oder sogar den CAN-Controller zuzugreifen. In den Modulen COEnvironment und COApplication werden die Einstellungen für die CANopen-Umgebung vorgenommen. Dies sind anwendungs- und CANopen-spezifische Programmteile. Aus diesem Grund werden sie nicht der reinen Anwendung zugeordnet. In diesen Modulen erfolgt u.a. indirekt die Auswahl des CAN-Controllers, der im Modul CANEnvironment initialisiert wird. Das hat den Vorteil, dass die Anwendung weiterhin unabhängig vom CAN-Controller ist.
Das Modul ***RemoteControl*** ist im Grunde genommen kein Bestandteil von CANopen. Es handelt sich um ein Steuerungsmodul vor allem für Debug-Zwecke, das zusätzlich hinzugelinkt werden kann. Unter Zuhilfenahme dieses Moduls können CANopen Geräte fernbedient werden (s. Kap. 4.11). Es besteht die Möglichkeit, z.B. über eine Console (s. Kap. 5), Kommandos an die Geräte zu schicken, deren Object Dictionary zu verändern oder sogar neue Programmdownloads durchzuführen. Für die Funktion dieses Moduls ist eine zusätzliche Kommunikation, z.B. per RS232 vorteilhaft. Prinzipiell ist aber auch eine Tunnelung über CANopen SDO's möglich.

4.3 Das CANopen Handle

Die in den Zielen genannte Unterstützung von mehreren CANopen-Instanzen erfordert ein modulares und flexibles Design der Software. Zu diesem Zweck wurde jeder Instanz eine Struktur, das ***CO_HANDLE*** zugeordnet. In dieser Struktur sind alle Informationen gespeichert, die nötig sind, um CANopen Objekte zu übertragen. Die Information des verwendeten CAN-Controllers befindet sich in dem Eintrag **CanHandle*. Das ist ein Zeiger auf ein CAN_HANDLE, der beim Senden von Messages an die CAN-API mitübergeben wird. Diese wählt aufgrund der gespeicherten Informationen den richtigen CAN-Controller aus. Der Eintrag **Objects* zeigt auf das verwendete Object Dirctionary, welches üblicherweise im Modul COEnvironment definiert sein sollte (s. Kap. 4.5). Den weitaus größten Teil des CO_HANDLE's machen die Callbacks aus. Das sind Funktionen, die bei Eintreten von gewissen Bedingungen aufgerufen werden, z.B. Eintreffen einer neuen Nachricht (s. Kap. 4.7).

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Declaration of CANopen handle
typedef struct
{
        CAN_MESSAGE         CanMessage;                 // current message buffer
        CAN_HANDLE          *CanHandle                  // can handle, length depends on controller type
        const OD_Object     *Objects;                   // ptr to object dictionary
        CO_CONNECTION       ClientConnection;           // used during SDO client transmissions
        CO_CONNECTION       ServerConnection;           // used during SDO server transmissions
        BYTE                ClientToggleBit:1;          // is there a client connection established (1 bit)
        BYTE                ServerToggleBit:1;          // is there a server connection established (1 bit)
        BYTE                NMTErrorControlToggleBit:1; // toggle bit for error control messages
        NODEID              NodeID;                     // NodeID of this client
        BYTE                Flags;                      // Flags described below
        BYTE                NMT_State;                  // store current NMT state
        CAN_Configcallback  CanConfigCallback;          // callback to initialize can controller
        void                *CanReceiveCallback;        // callback for received can messages
        void                *ServiceCallback;           // callback for CANopen callbacks (SDO finished...)
        void                *Userdata;                  // ptr to userdata, maybe instance to your window or task
        void                *InitializeCallback;        // callback for entering state NMT_INITIALISATION
        void                *PreOperationalCallback;    // callback for entering state NMT_PREOPERATIONAL
        void                *EndCallback;               // callback for closing connections
        void                *ReadRemoteObject;          // callback to read FPGA objects
        void                *InitWriteRemoteObject;     // callback to initialize FPGA access
        void                *ReadyRemoteObject;         // callback to free FPGA accesses
} CO_HANDLE;
```

Abbildung 4-2 Deklaration des CO_HANDLE

Durch die Benutzung des CO_HANDLE's können ein oder mehrere CAN-Instanzen gleichzeitig geöffnet werden. Dazu sind lediglich zwei initialisierte Strukturen nötig, die zwei verschiedene CAN-Controller benutzen.
Die Verwendung von CO_HANDLE's zwingt den Entwickler zusätzlich zu strukturierterem Programmieren und zu dem Verzicht, eine Vielzahl von Variablen *global* zu deklarieren. Dadurch wird die möglicherweise spätere Erweiterung von Anwendungen auf mehrere CANopen-Instanzen erleichtert.
Das Lesen bzw. Schreiben der Inhalte des CO_HANDLE's ist unzulässig. Die Anwendung darf die Zeiger auf das CO_HANDLE nur als Parameter an die Funktionen der CANopen-API übergeben. Eine Ausnahme bilden die Module *COEnvironment* und *COApplication*. In diesen Komponenten werden teilweise Einträge initialisiert, die für eine einwandfreie Arbeit nötig sind (s. Kap. 4.5).

4.4 Funktionen der CANopen API

Die in Kap. 2.7 beschriebenen Objekte und Funktionen sind lediglich der Rahmen für die allg. Implementierung. Zwischen CANopen-API und Anwendung erfolgt die Kommunikation jedoch über fest definierte Funktionen. Diese sind zur besseren Übersicht gestaffelt nach den Komponentennamen aufgelistet. Als erster Parameter wird grundsätzlich das CO_HANDLE als Zeiger übergeben.[7] Somit hat jede Funktion die Möglichkeit, auf die Attribute der jeweilige CANopen-Instanz zuzugreifen.
Die Namen der Funktionen wurden möglichst selbsterklärend gewählt und enthalten in den meisten Fällen einen Hinweis auf das entsprechende Funktionsmodul.

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// CANopen Main functions
OPTOERROR       CO_InitCANopen(CO_HANDLE *handle, CO_Configcallback configcallback );
OPTOERROR       CO_EndCANopen(CO_HANDLE *handle);
void            CO_MessageLoop(CO_HANDLE *handle);
OPTOERROR       CO_WaitServices(CO_HANDLE *handle, long milliseconds);
OPTOERROR       CO_SetBaudrate(CO_HANDLE *handle, int baudrate);
```

Abbildung 4-3 Deklaration der CANopen Hauptfunktionen

Die Hauptfunktionen der CANopen-API dienen zum Auf- und Abbau von CANopen-Instanzen (CO_InitCANopen(), CO_EndCANopen()) und dem Behandeln von empfangenen Nachrichten (CO_MessageLoop()).
Mit der Funktion CO_WaitServices() wird auf das Ende aller initiierten Nachrichtenübertragungen gewartet. Dies ist notwendig, da CAN-Controller intern Messagequeues besitzen und nicht garantiert werden kann, dass eine Nachricht gesendet wurde, wenn die entsprechende Hardware-Funktion der CAN-API erfolgreich beendet wurde. Auf eine Implementierung der Wait-Funktion in den Hardware-Treibern wurde verzichtet, da dann die Gefahr eines Deadlocks durch blockierende Stationen am Bus bestünde.
Die Funktion CO_SetBaudrate() dient dem Setzen der Baudrate des Gerätes. Zu beachten ist in diesem Zusammenhang, dass dadurch die Baudraten der anderen Geräte nicht verändert werden. Es ist demnach vorteilhaft, einen Mechanismus in die Anwendung einzubinden, der die Geräte synchronisiert (s. Kap. 4.5). Die voreingestellte Baudrate beträgt 100 KBit/s.

[7] Entspricht in C++ dem this-Zeiger.

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Function declarations for service data objects (SDO)
OPTOERROR           SDO_Download(CO_HANDLE *handle, CO_CONNECTION *connection, NODEID nodeid);
OPTOERROR           SDO_Upload(CO_HANDLE *handle, CO_CONNECTION *connection, NODEID nodeid);
```

Abbildung 4-4 CANopen Funktionen für Service Data Objects (SDO)

Die Anwendung erhält mit den Funktionen SDO_Download() und SDO_Upload() die Möglichkeit, komplexe Datenobjekte zu senden bzw. anzufordern. Der Parameter CO_CONNECTION entspricht dabei einer Struktur, in der die Multiplexoren der Daten des eigenen und des fremden Object Dictionaries abgelegt sind. In dem Argument NODEID ist die Nummer des Gerätes definiert, mit dem kommuniziert werden soll. Ein Broadcast (NODEID=0) ist nicht zulässig.
Da sich hinter diesen Funktionen immer Hand-Shake Verfahren verbergen, ist es ratsam, sie nicht in echtzeit-relevanten Abschnitten zu verwenden. Die Funktionen kehren erst bei erfolgreichem Abschluß der Kommunikation bzw. bei Auftreten eines Fehler, z.B. Timeout, wieder zurück. Der Wert des Timeouts ist standardmäßig auf 300ms festgelegt. Er kann aber in der Initialisierungsphase für jede Instanz geändert werden.

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Function declarations for process data objects (PDO)
OPTOERROR           PDO_WritePDO(CO_HANDLE *handle, PDO_NUMBER pdonumber);
OPTOERROR           PDO_ReadPDO(CO_HANDLE *handle, PDO_NUMBER pdonumber);
```

Abbildung 4-5 CANopen Funktionen für Process Data Objects (PDO)

Zur Echtzeitkommunikation erhält die Anwendung mit den Funktionen PDO_WritePDO() und PDO_ReadPDO() die Möglichkeit, Daten schnell auszutauschen. Das zweite Argument bestimmt die PDO-Nummer. Jedes Gerät kann im pre-defined connection set 4 TPDO's und 4 RPDO's anfordern, d.h. aus dem Bereich der 512 möglichen Nummern sind nur 8 gültig. Die Berechnung der PDO-Nummern ist in Kap. 2.7.1 erläutert. Für jede PDO-Nummer müssen im Object Dictionary die Einträge für Mapping und Communication vorhanden sein, sonst kehrt die Routine mit einem Fehler zurück. Die Kenntnis der PDO-Nummer setzt voraus, dass ebenfalls Kenntnis über die Art der zu übertragenen Daten besteht, sodass das Mapping nicht wie in CANopen vorgesehen, durch die Applikation erfolgen muß, sondern in den CANopen-API Funktionen vorgenommen werden kann – eine deutliche Entlastung für Anwendungen. Zu beachten ist in diesem Zusammenhang, dass die Funktion PDO_ReadPDO() ein Handshake durchführt.

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// function declarations for network management (NMT)
OPTOERROR           NMT_SetState(CO_HANDLE *handle, BYTE command);
OPTOERROR           NMT_SendRequest(CO_HANDLE *handle, NODEID nodeid, BYTE command );
```

Abbildung 4-6 CANopen Funktionen für das Network Management (NMT)

Für das Netzwerkmanagement stehen zwei Funktionen NMT_SetState() und NMT_SendRequest() zur Verfügung. Mit NMT_SetState() wird der interne Zustand der CANopen Software für diese Instanz gesetzt. Dies ist allerdings nur in den seltensten Fällen nötig, da das Netzwerkmanagement vom NMT-Master übernommen wird. Dieser ist in der Lage, NMT Übertragungen zu initiieren. Dazu wird die Funktion NMT_SendRequest() benutzt. Die Node-ID kennzeichnet das Gerät, an das das NMT-Object geschickt werden soll. Eine Broadcast (No-

de-ID=0) ist hier grundsätzlich möglich. Dadurch besteht die Möglichkeit, das komplette Netzwerk vom Status NMT_PRE-OPERATIONAL gleichzeitig in den Modus NMT_OPERATIONAL zu setzen.
Das Benutzen dieser Funktion ist nur zulässig, wenn das Gerät der NMT_Master des Netzwerkes ist.

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// function declarations for network management error control(NMTEC)
OPTOERROR           NMTEC_SendLifeguardRequest(CO_HANDLE *handle, NMTEC_INFO *nmtec_info);
OPTOERROR           NMTEC_SendHeartbeatRequest(CO_HANDLE *handle);
```

Abbildung 4-7 Funktionen zum Network Management Error Control

Eine Untergruppe der Funktionen des Network Managements stellen die *Network Management Error Control* Services dar. Sie werden benutzt, um im laufendem Betrieb sicherzustellen, dass Ausfälle von Geräten eindeutig erkannt werden und bei Problemen die CANopen-Applikation sicher heruntergefahren werden kann.
Die in Abbildung 4-7 dargestellten Funktionen bieten die Schnittstellen zu den in Kap. 2.8 erläuterten Überwachungsmechanismen *Lifeguarding* und *Heartbeat*. Bei beiden Verfahren werden Timer zur Überwachung und Sendung der jeweilen Anfragen benötigt. Die Timer sind allerdings nicht Bestandteil der CANopen Software, um die Plattformunabhängigkeit nicht zu gefährden.

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// function declarations for sync objects
OPTOERROR           SYNC_SendObject(CO_HANDLE *handle);
```

Abbildung 4-8 Funktionen des SYNC-Objects

Die Synchronisation von Teilnehmern erfolgt durch periodisches Senden des SYNC-Objects durch den SYNC-Master. Dies kann ein beliebiges Gerät im CANopen Netzwerk sein und ist nicht an die NMT-Master Funktionalität gebunden. Das Einhalten von Zeitgrenzen bleibt dabei Aufgabe der Anwendung. Die CANopen-API stellt lediglich die Funktion zur Verfügung, das SYNC-Object zu senden. In den Netzwerkteilnehmern, die diese Nachricht auswerten, wird ein Callback ausgelöst, indem dann synchronisierte Aktionen durchgeführt werden können.

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// function declarations to control CANopen devices remotely
void                CO_RemoteCommand(CO_HANDLE *handle, const char *command);
```

Abbildung 4-9 CANopen Funktionen für die Fernwartung (Remote Control)

Um ein Gerät fernzuwarten, muss die Funktion CO_RemoteCommand() implementiert sein. Dies ist standardmäßig nicht der Fall. Eine umfassende Beschreibung dieser Funktion befindet sich in Kap. 4.11.

4.5 CANopen Application

Die CANopen Application (kurz ***COApplication***) ist ein Funktionsmodul, dass in allen Geräten desselben Netzwerkes gleich ist. Es handelt sich um globale Einstellungen, die in jedem Gerät gelten. Die Umsetzung dieser Softwarekomponente ist sehr anwendungsspezifisch und kann nur allgemein erklärt werden.

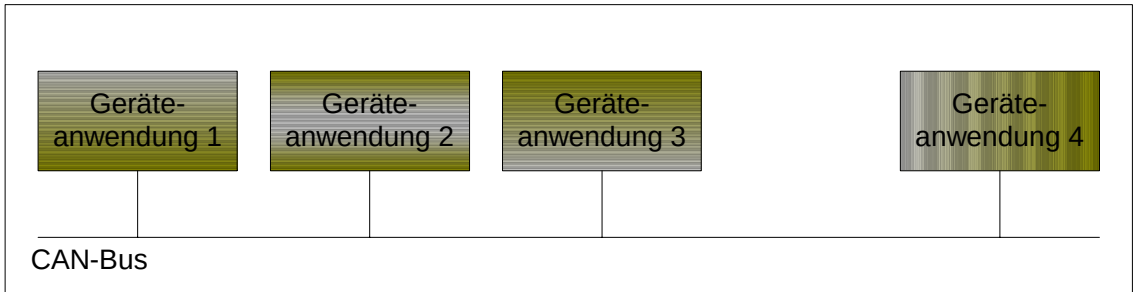

Abbildung 4-10 Geräteanwendung vs. CANopen Anwendung

In Abbildung 4-10 ist das grundsätzliche Modell einer ***CANopen Anwendung*** bzw. ***CANopen Application*** gezeigt. Eine Anwendung könnte sein, die Temperatur in einem Gebiet zu messen und die Messergebnisse auszuwerten. Gerät 1 nimmt die Temperatur an Ort 1, Gerät 2 an Ort 2 und Gerät 3 an Ort 3 auf. Alle 3 übertragen ihre Meßwerte an Gerät 4, welches die Daten auswertet und an den übergeordneten Prozess weiterleitet. Jedes Gerät besitzt trotz der ähnlichen Aufgaben seine Spezifika, z.B. seine eigene Node-ID, eventuell eigene Messmethoden oder sogar unterschiedliche Hardware. Man spricht von einer ***Geräteanwendung*** bzw. ***device application***. Alle Geräteanwendungen zusammen bilden die CANopen Anwendung. Eine CANopen Anwendung umfaßt mindestens 2 Geräteanwendungen. Komponenten, die in allen Geräten gleich sind, werden in dem Modul COApplication definiert.
Um in einer Geräteanwendung ein Objekt eines fremden Gerätes zu herunterzuladen, ist die Kenntnis des entsprechenden Multiplexors notwendig. Aus diesem Grund sind die Indizies aller Object Dictionaries in diesem Modul deklariert. Vorteilhaft ist die COApplication auch für Debug-Zwecke, um z.B. Ausgaben nach *stdout* in einem konstanten Format zu machen, die später durch einen *Parser* gefiltert werden können.

4.6 CANopen Environment

Das CANopen Environment (kurz COEnvironment) bildet die Schnittmenge zwischen Geräteanwendung, Anwendung, CANopen und dem Betriebssystem. Es handelt sich vorwiegend um geräteabhängige Einstellungen zur Systemumgebung und Behandlung von Ereignissen, die im Zusammenspiel mit CANopen auftreten können. Die Umsetzung dieser Softwarekomponente ist ebenfalls sehr spezifisch und kann nur allg. erklärt werden. Sie wird für jedes Gerät extra definiert. Zu den Aufgaben dieses Softwaremodul gehören:

- Initialisierung und Zerstörung der CANopen Instanz
- Anlegen des gerätespezifischen Object Dictionaries
- Behandeln von Callbacks
- Synchronisation der Netzwerkteilnehmer
- Netzwerkmanagement
- Interaktion mit angekoppelter Hardware

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////
// This is a callback function and will be called during InitCANopen().
// Here you are able to set some parameters to create the CANopen
// instance successfully
OPTOERROR APP_ConfigureCANopen(CO_HANDLE *cohandle)
{
        //TRACE("APP_ConfigureCANopen entered\n");

        cohandle->CanConfigCallback      = ENV_CAN_ConfigureHardware;
        cohandle->ServiceCallback        = (void*)APP_ReceiveCallback;
        cohandle->CanReceiveCallback     = (void*)APP_CANReceiveCallback;
        cohandle->InitializeCallback     = (void*)APP_Initialize;
        cohandle->PreOperationalCallback = (void*)APP_PreOperational;
        cohandle->EndCallback                    = (void*)APP_End;
        cohandle->ReadRemoteObject       = 0;
        cohandle->InitWriteRemoteObject  = 0;
        cohandle->ReadyRemoteObject      = 0;
        cohandle->Objects                = OD;                  // set object dictionary
        cohandle->Flags                  = DEVICE_FLAGS;        // set flags (e.g.
NMT_MASTER)
        cohandle->NodeID                 = DEVICE_COBID;        // set device ID
        return NOERROR;
}
```

Abbildung 4-11 Beispiel einer CANopen Initialisierung

In Abbildung 4-11 ist der grundsätzliche Ablauf einer CANopen Initialisierungsroutine gezeigt. Bevor diese Funktion aufgerufen wird, ist das CO_HANDLE mit Standardwerten gefüllt, sodass hier nur noch die abweichende Initialisierung erfolgen muss. Das derzeit auskommentierte TRACE Kommando dient normalerweise zur Ausgabe einer Meldung auf *stdout*, um den Fortschritt bei der Initialisierung zu überwachen. Die nächsten 9 Anweisungen dienen dem Setzen der Callback-Routinen. Da diese im allg. gerätespezifisch sind, sind sie ebenfalls im COEnvironment definiert. In dem gezeigten Beispiel werden Callbacks aufgerufen, wenn

- ein CAN-Controller initialisiert wird
- eine CANopen Funktion, z.B SDO-Download, PDO_Read abgeschlossen bzw. angefordert wird
- eine CAN-Message empfangen wurde
- der NMT_State auf NMT_INITIALISATION gesetzt wird
- der NMT_State auf NMT_PRE-OPERATIONAL gesetzt wird
- die CANopen Instanz beendet wird

Die Anweisung *cohandle->Object=OD* setzt den Zeiger auf das Object Dictionary, das hier als konstantes, globales Array in dem Modul COEnvironment angelegt wurde. Dies ist durchaus legitim, da das Object Dictionary konstant ist und über die Funktion APP_ConfigureCANopen() jederzeit die Möglichkeit besteht, ein anderes Array für eine weitere Instanz festzulegen. Durch die flexible Verkettung wird die in Kap. 4.1 einfache Erweiterbarkeit der CANopen-API erreicht.
Die Device-Flags legen den Typ des Gerätes fest, z.B. ob es ein NMT_Master oder NMT_Slave ist. Als letztes wird die Node-ID gesetzt, die das Gerät im Netzwerk eindeutig identifiziert.

4.7 Callback-Funktionen

In Abbildung 4-12 sind die möglichen Funktionen/Callbacks für das Modul COEnvironment bzw. COApplication gezeigt. Die Callbacks sind nicht alle zwingend erforderlich. Dem Ent-

wickler steht es offen, sie zu benutzen. Die Callbacks werden einmalig in der Funktion APP_ConfigureCANopen() gesetzt. Die Namen der Funktionen sind Richtwerte, trotzdem sollte die Namenskonvention eingehalten werden, um die Lesbarkeit des Programms zu verbessern und zu vereinheitlichen.

```
void        APP_Initialize(CO_HANDLE *cohandle)
OPTOERROR   APP_PreOperational(CO_HANDLE *cohandle)
void        APP_ReceiveCallback(CO_HANDLE *handle, WORD event, void *data)
OPTOERROR   APP_Reset(CO_HANDLE *cohandle)
OPTOERROR   APP_ReadRemoteObject(CO_HANDLE *handle, MULTIPLEXOR multiplexor, BYTE **data, WORD *size)
OPTOERROR   APP_InitWriteRemoteObject(CO_HANDLE *handle, MULTIPLEXOR multiplexor, BYTE **data, WORD *size)
OPTOERROR   APP_ReadyRemoteObject(CO_HANDLE *handle, MULTIPLEXOR multiplexor, int command)
```

Abbildung 4-12 Callback-Funktionen im Modul COEnvironment

Die Funktion *APP_Initialize()* wird aufgerufen, wenn die CANopen-API in den Status NMT_INITIALISATION wechselt. Dies kann durch Reset oder ein externes NMT-Kommando erfolgen.
Hier sollte eine grundlegende Initialisierung aller Variablen erfolgen. Die Initialisierung des Object Dictionaries muß in dieser Funktion abgeschlossen werden – wichtig vor allem bei Array's (s. Kap. 4.9). Das Setzen der Start-Baudrate muß hier erfolgen, falls diese von der Standardbaudrate abweicht. Ein Zugriff auf das Netzwerk per SDO's oder PDO's ist nicht erlaubt.
Nach Abschluß der Routine wird die CANopen-Software automatisch in den Status NMT_PRE-OPERATIONAL gesetzt.

In dem Callback *APP_PreOperational()*, der bei Eintritt in den Status NMT_PRE-OPERATIONAL nach Senden des *BootupEvents* aufgerufen wird, erfolgt die Aktualisierung der Geräteanwendung zur Laufzeit, z.B. durch Laden der aktuellen Systemzeit. Weiterhin ist es möglich, umfangreiche Selbsttests durchzuführen.
Eine Änderung der Baudrate durch den NMT_Master sollte hier erfolgen. Die Synchronisation der Geräte bleibt dem Entwickler überlassen. Zu empfehlen ist, einen Eintrag im Object Dictionary für die Baudrate zu reservieren und diese dann beim Austritt aus der Funktion zu setzen.
Der Returnwert dieser Funktion sollte möglichst Null sein, ansonsten wird bei dem ersten fehlerhaften Aufruf des Callbacks die CANopen Instanz geschlossen.

Die Benachrichtigung über Ereignisse innerhalb der CANopen-API erfolgt durch Aufruf der Callback-Funktion *APP_ReceiveCallback()*. Der Parameter *Event* beinhaltet die Nummer des Ereignisses. Das dritte Argument **data* ist ein Zeiger auf ein Datenelement, das abhängig vom Ereignistyp ist. Derzeit sind die folgenden Ereignisse definiert:

- SDO_TX_DOWNLOAD — Informiert über den erfolgreichen Abschluß eines selbst initiierten SDO-Downloads. Der Parameter **data* zeigt auf den Multiplexor des Objektes im geräte-eigenen Object Dictionary.
- SDO_TX_UPLOAD — Informiert über den erfolgreichen Abschluß eines selbst initiierten SDO-Uploads. Der Parameter **data* zeigt auf den Multiplexor des Objektes im geräte-eigenen Object Dictionary.
- SDO_RX_DOWNLOAD — Informiert über den erfolgreichen Abschluß eines vom Netzwerk initiierten SDO-Downloads. Der Parameter **data* zeigt auf den Multiplexor des Objektes im geräte-eigenen Object Dictionary.

- SDO_RX_UPLOAD — Informiert über den erfolgreichen Abschluß eines vom Netzwerk initiierten SDO-Uploads. Der Parameter **data* zeigt auf den Multiplexor des Objektes im geräte-eigenen Object Dictionary.
- NMT — NMT_State geändert. Der Parameter **data* ist nicht definiert.
- SYNC — Sync-Object empfangen. Der Parameter **data* ist nicht definiert.
- PDO_TRANSMISSION — Informiert über Abschluß einer PDO Übertragung. Der Parameter **data* zeigt auf die Struktur PDO_Result, in der das Ergebnis der Operation gespeichert ist.
- PDO_RECEPTION — Eine PDO Übertragung von einem anderen Gerät wurde empfangen. Der Parameter **data* zeigt auf die Struktur PDO_Result, in der das Ergebnis der Operation gespeichert ist.
- PDO_REQUEST — Ein PDO Request wurde von einem anderen Gerät empfangen. Die Anwendung kann jetzt die erforderlichen Daten, die in den entsprechende Mapping-Einträge spezifiziert sind, für dieses PDO zusammenstellen. Nach Abschluß des Callbacks wird der PDO-Request beantwortet. Der Parameter **data* zeigt auf die Struktur PDO_Result, in der das Ergebnis der Operation gespeichert ist.

Die Funktion *APP_Reset()* dient zum Rücksetzen des kompletten Gerätes. Dies ist vor allem während einer Fernwartung von Vorteil, um z.B. das Netzwerk komplett neu zu booten oder einen Download zu starten.

4.8 Remote Objects

Die Callback-Funktionen *ReadRemoteObject()*, *ReadyRemoteObject()* und *InitWriteRemoteObject()* aus Abbildung 4-12 dienen dem Zugriff auf angekoppelte Hardware. Dies ist notwendig, um Einträge im Object Dictionary zu adressieren, die nicht in den Speicher eingeblendet werden können, z.B. Register in FPGA-Bauelementen. Diese Objekte heißen ***Entfernte Objekte*** bzw. ***remote objects***.
Die Funktion ReadRemoteObject() wird benutzt, um einen Lesezugriff zu starten. Dabei wird in dem Callback der Inhalt des entfernten Registers in einen temporären Speicher kopiert. Die Adresse und die Größe des benutzten Speichers werden in den übergebenen Argumenten gespeichert. Die CANopen Software benutzt nach Verlassen der Callback-Funktion den temporären Speicher wie einen normalen Objektspeicher. Das Verhalten ist völlig transparent. Ist der Zugriff auf das Objekt beendet, wird der Callback ReadyRemoteObject() aufgerufen. Hier besteht nun die Möglichkeit, das entfernte Objekt freizugeben. Das Verhalten der Geräteanwendung zwischen den beiden Callbacks ist nicht definiert.
Im Falle eines Write-Zugriffs auf ein entferntes Objekt wird der Callback InitWriteRemoteObject() aufgerufen. Hier besteht nun die Möglichkeit, das Objekt zu blockieren. Innerhalb dieser Funktion müssen ebenfalls die temporären Speicher gesetzt werden. Die CANopen-API benutzt diesen Speicher zum Ablegen der Daten. Nach Abschluß des Schreibvorgangs wird die Funktion ReadyRemoteObject() aufgerufen. Hier erfolgt nun das Schreiben der Daten in das entfernte Objekt.

Im Object Dictionary kann für jedes Objekt bereits im Vorfeld temporärer Speicher zugeordnet werden. In den meisten Fällen ist es dann ausreichend, für Read-Zugriffe die Funktion ReadRemoteObject() und für Write-Zugriffe die Funktion ReadyRemoteObject() aufzuwerten.

4.9 Aufbau des Object Dictionaries

Der Aufbau des Object Dictionaries (***OD***) innerhalb der CANopen-API erfolgt durch Array's. Durch die in Kap. 2.4 erwähnten Nachteile entspricht der Index eines Elements in diesem Array nicht dem Index des Objekts im Object Dictionary. Viel mehr muß nach den Einträgen in dem Array gesucht werden, d.h. Element für Element wird mit dem MUX verglichen. Das kostet sehr viel Prozessorzeit. Bei performance-schwachen Systemen sollte daher das Object Dictionary möglichst knapp gehalten sein. Die kritischten Einträge, etwa PDO-Mapping und PDO-Communication Parameter sollten am Anfang des OD stehen.

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// declaration of one entry of the object dictionary
typedef struct
{
        WORD    Index;          // Index of object
        BYTE    ObjectType;     // classification of object (record, var, array)
        OD_Entry Entry;         // Entry structure
} OD_Object;
```

Abbildung 4-13 Deklaration eines Objekts im Object Dictionary

Jedem Objekt wird in diesem Array eine ***OD_Object*** Struktur zugewiesen (Abbildung 4-13). Der 16-Bit breite *Index* beinhaltet den Index des Multiplexors dieses Objekts. Bei einem Zugriff auf ein Objekt wird als erstes das gesamte Object Dictionary nach dem passendem Index durchsucht. Ein Index von null ist das Kennzeichen für ein Ende des Object Dictionaries. Das Element *ObjectType* spezifiziert den Typ des Objekts. Folgende Typen werden unterstützt:

- OD_VARIABLE — Objekt besteht aus einem einfachen Datentyp (z.B. Integer8)
- OD_ARRAY — Objekt besteht aus einer Sammlung gleicher Datentypen (z.B. Vektor, String). Das erste Element ist von Typ Integer8 und enthält die Anzahl der Elemente.
- OD_RECORD — Objekt besteht aus einer Sammlung unterschiedlicher Datentypen. Das erste Element ist von Typ Integer8 und enthält die Anzahl der Elemente.
- OD_DOMAIN — Objekt ist nicht weiter spezifiziert. Die Länge ist variabel.
- OD_DEFSTRUCT — Objekt besteht aus einer vordefinierten Sammlung unterschiedlicher Datentypen (z.B. Time_of_Day). Das erste Element ist von Typ Integer8 und enthält die Anzahl der Elemente.

Das Element *Entry* in der *OD_Object* Struktur beinhaltet eine ***OD_Entry*** Struktur. Der Inhalt ist abhängig vom *ObjectType* (s. Abbildung 4-14).

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////
// declaration of one entry in records and single types
typedef struct
{
        BYTE    Flags;          // currently in use (downloading, uploading, changing...)
                                // some flags are not used because OD_Objects must be declared as
                                // const to work correctly in conjunction with Keil C Compiler
        BYTE    Type;           // boolean, float, int, uint time_of_day
        BYTE    Attributs;      // read/write, read only, const, write only
        OD_Data Data;           // if variable then ptr to variable
                                // if array or record then ptr to array of OD_Entry
                                // which points to the data
} OD_Entry;
```

Abbildung 4-14 Deklaration eines Sub-Elements in einem Objekt

Die Elemente der *OD_Entry* Struktur haben folgende Bedeutung:

- *Flags* Die Flags geben Auskunft darüber, in welchem Speicherbereich die Daten *(*Data*)* abgelegt wurden. Folgende Speicherflags sind möglich:
  - ODF_RAM Daten, auf die *Data* zeigt, werden im RAM abgelegt.
  - ODF_ROM Daten, auf die *Data* zeigt, werden im ROM abgelegt.
  - ODF_ROMD In diesem Fall repräsentiert *Data* bereits die Daten. Es gibt keinen Zeiger. Nur zulässig für Variablen bis zu 32 Bit.
  - ODF_FPGA Kennzeichnet ein *entferntes Objekt*. Bei jedem Zugriff werden die Callback-Funktionen ReadRemoteObject() und InitWriteRemoteObject() aufgerufen.

  Die Festlegung der Speicherbereiche ist unabhängig von der Definition der Zugriffsrechte, die im Element Attributes definiert sind, d.h. ein Datum kann auch als *const* bzw. *read only* deklariert werden, wenn es im RAM liegt.
  Ursprünglich war vorgesehen, einen Locking-Mechanismus einzubauen, der das gleichzeitige Schreiben vom Netzwerk und der Anwendung aus verhindert. Da der Keil-Compiler nicht in der Lage war, ein Array anzulegen, das nicht als ***const*** deklariert ist, konnte die Information nicht gespeichert werden.[8]
- *Type* Hier wird der einfache Datentyp festgelegt, auf den in *Data* verwiesen wird. Abhängig von *ObjectType* gibt es 2 Bedeutungen:
  - Spezifiziert der *ObjectType* eine Variable (*OD_Variable*) oder eine Domain (*OD_Domain*), legt dieser Eintrag den einfachen Datentyp fest, auf den Data zeigt. Einfache Datentypen sind u.a. Integer8, VisibleString und Boolean.
  - In allen anderen Fällen ist der Typ nicht gesetzt, da Data auf ein Array von *OD_Entries* zeigt.
- *Attributes* Legt die Zugriffsrechte dieses Elements für das Netzwerk fest. Es sind folgende Attribute möglich:
  - ReadOnly
  - WriteOnly
  - Const
  - Void
  - ReadWrite

[8] Die Definition eines Array's im Keil-Compiler hat ungeahnte Nebeneffekte. So erzeugt folgende Schreibweise ein Array lauter Nullen: OD_Object OD[]={ { COID1_SCAN , OD_DOMAIN , ODF_RAM, ODDT_VISIBLE_STRING , 0 , BPTR(Domain_Scan) }, 0 };
Definiert man dasselbe Array als const, wird es korrekt initialisiert, landet dafür im ROM und kann nicht mehr geändert werden. Warnungen werden natürlich unterschlagen.

- *Data* Data ist ein 32-Bit Datum, das als Union vom Typ *OD_Data* definiert wurde. Die Deklaration von *OD_Data* sieht wie folgt aus:

```
typedef union
{
        const UINT32   Data;      // \ do not change the order. Keil compiler
        const void     *DataPtr;  // / can only handle this order[9]
} OD_Data;
```

  Daraus folgt, dass das Datum in *Data* entweder ein Datum vom Typ *UINT32* oder ein Zeiger vom Typ *void* ist. Die Festlegung ist abhängig von *Type* und von *ObjectType*.
  Im Normalfall wird *Data* als Zeiger benutzt (*Data.DataPtr*). Es besteht aber auch die Möglichkeit, dass *Data* als Speicher für das Datum benutzt wird (*Data.Data*). Dann sind allerdings nur Datentypen erlaubt, die max. 32-Bit breit und konstant sind, denn das Object Dictionary liegt in konstanten Speicherbereichen (ROM). Der Vorteil der Union besteht in der Einsparung eines Zeigers, was zusätzlich die Lesbarkeit des Quellcodes erhöht.
  Eine Erweiterung auf 64-Bit Zeiger ist jederzeit möglich. Aus Kompatibilitätszwecken dürfen allerdings nur Datentypen bis zu einer Breite von 32-Bit verwendet werden.

```
const OD_Object OD[]=
{
{ ID_DEVNUMBER , OD_VARIABLE , ODF_ROMD , ODDT_INTEGER32 , ODA_CONST      , BINT(DEVNUMBER) },
{ ID_DEVID     , OD_VARIABLE , ODF_ROM  , ODDT_STRING    , ODA_READONLY , BPTR(DEVID)     },
{ ID_SCAN      , OD_DOMAIN   , ODF_RAM  , ODDT_STRING    , 0            , BPTR(Scan)      },
{ ID_CURSCAN   , OD_ARRAY    , ODF_FPGA , ODDT_INTEGER8  , 0            , BPTR(CurBuffer) },
{ ID_PDO1      , OD_RECORD   , ODF_ROM  , 0              , 0            , BPTR(Mapping1)  },
0
};
```

Abbildung 4-15 Beispiel eines Object Dictionaries

In Abbildung 4-15 ist die verkürzte und vereinfachte Variante eines Object Dictionaries gezeigt. Es besteht aus 5 Objekten. Der erste Eintrag in der Zeile steht für den Index des Elements, gefolgt vom *ObjectType*. Danach werden die vier Unterelemente von der Struktur *OD_Entry* (s. Abbildung 4-13) gesetzt.
In der ersten Zeile wird ein konstantes Objekt vom Typ Integer32 definiert. Dies ist ein 32-Bit Wert, der direkt nach *Entry.Data.Data* geschrieben wird, d.h. für dieses Element wird kein Zeiger benutzt. Zum Setzen des richtigen Eintrages in der Union *OD_Entry* wird das Macro ***BINT()*** benutzt. Es wandelt einen Wert so um, dass es vom Compiler richtig in die *OD_Entry.Data.Data* Struktur eingesetzt wird. Wichtig in diesem Zusammenhang ist, dass der Speichertyp auf *ODF_ROMD* steht, ansonsten erkennt die CANopen-Software später nicht, dass in Entry.Data ein Wert und kein Zeiger steht.

Der zweite Eintrag des OD aus Abbildung 4-15 legt eine Variable an, die in Wirklichkeit ein String ist. Die Buchstabenbreite beträgt 8 Bit, wovon laut Defintion nur 7 Bit benutzt werden

[9] Wenn die Reihenfolge dieser Union geändert wird, gibt der Keil-Compiler eigenartigerweise unendlich viele Warnungen aus.

dürfen. Dieser String ist *readonly*. Durch das Macro ***BPTR()*** wird die Adresse des Strings im Speicher in den Eintrag *Entry.Data.DataPtr* geschrieben. Zu beachten ist hier, dass der Speichertyp nicht ODF_ROMD sein darf, da es sich nicht um einen Wert handelt.
Strings werden in dieser CANopen Software bisher nicht nach CANopen Standard behandelt. Der Standard verlangt Strings so abzulegen, dass erst ein Byte mit der Längenangabe erfolgt und danach der String ohne Terminierungszeichen (Null). Das führt in der Praxis zu erheblichen Schwierigkeiten:

- Dieses Format wird von Compilern nicht unterstützt, was zur Folge hat, das die Strings zur Laufzeit zusammengesetzt werden müssen. Dazu benötigt man doppelten Speicher (ROM zum Ablegen der Strings, RAM um den CANopenString abzulegen).
- Ein String kann nicht länger als 256 Zeichen sein
- Dieses Format ist nicht kompatibel mit gängigen Standards, z.B. dem der ANSI-C Funktionen printf() und open(). Bei einem Aufruf dieser Funktionen müßten die Strings wiederum konvertiert werden.
- Diese Definition verleitet den Entwickler zu statischen Programmen, indem er komplette Strings in Array's Buchstabe für Buchstabe eingibt. Eine schnelle Änderung von Strings ist nicht möglich. Die Längenangabe korreliert nicht mit den Daten, da sie nicht automatisch berechnet wird.
  Bsp: char devicename[]={ 8, 'D','e', 'v', 'i', 'c', 'e', ' ', '1'};

Um die Einbindung der CANopen Anwendung in bestehende Netzwerke nicht unnötig zu erschweren, kann die Software aber ohne große Probleme auf die im Standard beschriebene Version umgestellt werden. Dazu sind nur einige Änderungen im Modul *CANopen/CO_ObjectDictionary.c* nötig. ANSI-C Strings könnten in herstellereigenen Anwendungen auch als Variablen vom Typ *OD_Domain* definiert werden.

Der dritte Eintrag aus dem OD in Abbildung 4-15 ist eine Domain, d.h. CANopen kennt den Typ der Variable nicht und kann sie daher nur als reinen Speicher ansprechen. BPTR(Scan) legt einen Zeiger auf den Speicherbereich fest. Um die genaue Datenlänge zu ermitteln, wird bei jedem Zugriff auf dieses Objekt die Funktion ReadRemoteObject() aufgerufen.

Das vierte Objekt ist ein Array aus 8-Bit Werten. Dieses Array liegt im FPGA, d.h. es ist ein *entferntes Objekt*. Daher wird bei jedem Zugriff die Funktion ReadRemoteObject() bzw. InitWriteRemoteObject() aufgerufen. In diesen Routinen können die Daten vom FPGA geladen bzw. in den FPGA gespeichert werden. Zu beachten ist in diesem Zusammenhang, dass das Format des Array's stimmen muss, d.h. der erste Eintrag bestimmt die Anzahl der Elemente. Er darf nicht unterschlagen werden. Durch die Definition des temporären Speichers CurBuffer braucht der Callback nur noch die Daten in diesen Buffer zu kopieren und die Länge richtig zu setzen.

Das 5. Element in Abbildung 4-15 definiert ein *Record*, d.h es handelt sich um eine Struktur, die aus mehreren unterschiedlichen Elementardatentypen besteht. Im Beispiel wird das Mapping eines PDO's dargestellt. Aus diesem Grund ist *Data.Dataptr* hier ein Zeiger auf Mapping1 – eine Struktur, die in Abbildung 4-16 gezeigt ist.
Der erste Eintrag in der Struktur ist vom Typ Integer8 und definiert die Anzahl der weiteren Unterelemente – in diesem Fall zwei. Jeder folgende Eintrag beinhaltet einen Mapping-Eintrag. Es handelt sich dabei um 32-Bit Werte, die im höherwertigen 16-Bit Teil den Wert des Indizies des zu mappenden MUX's enthalten. Der niederwertige Teil besteht aus einem 8-Bit Wert für den SubIndex und einem 8-Bit Wert, der die Länge der zu übertragenen Daten dieses Objekts in Bit angibt. In dem dargestellten Beispiel wird demnach das Objekt [ID_NUMEDGES][0] mit einer Länge von 8 Bit=1 Byte und danach das Objekt [ID_EDGES][0] mit einer Länge von 56

Bit=7 Byte gemappt. Zusammen entspricht das einer Datenlänge von 8 Bytes, was genau dem max. Volumen einer PDO-Nachricht genügt. Das Format der PDO-Nachricht muß allen PDO-Consumern bekannt sein, da sie sonst den Inhalt der Nachricht nicht entschlüsseln können.

```
const OD_Entry Mapping1[]=
{
{ ODF_ROMD      , ODDT_INTEGER8    , ODA_READONLY    , BINT(2)                              },
{ ODF_ROMD      , ODDT_INTEGER32   , ODA_READONLY    , BINT( (ID_NUMEDGES*65536) | 8 )      },
{ ODF_ROMD      , ODDT_INTEGER32   , ODA_READONLY    , BINT( (ID_EDGES*65536) | 56 )[10]    },
0
};
```

Abbildung 4-16 Beispiel eines OD_Records anhand eines PDO-Mappings

Bei der Arbeit mit Object Dictionaries ist weiterhin zu beachten, dass ein Zugriff auf komplexe Strukturen mit einem SubIndex von Null immer die gesamte Struktur adressiert. Daraus folgt, dass die Daten im Speicher hintereinander liegen müssen. Eine Speicherung von Elementardatentypen mittels ODF_ROMD und BINT(), wie in Abbildung 4-16 gezeigt, ist dann nicht zulässig. Der Entwickler kann entscheiden, ob es nötig ist, die Daten exakt hintereinander ablegen zu müssen. Dabei ist zu beachten, dass manche Compiler sog. Alignments durchführen, d.h. sie legen jeden neuen Datentyp auf eine gerade Adresse. Andere Compiler, z.B. der Keil-Compiler, bieten Optionen an, um die Datenreihenfolge zu ändern.

Für das Anlegen von eigenen Object Dictionaries sollte der Entwickler folgende Punkte unbedingt beachten, da sich eine Fehlersuche erfahrungsgemäß als äußerst schwierig und langwierig erweist:

- Sind alle Einträge richtig und sinnvoll ?
- Ist die Suchreihenfolge eingehalten, d.h. sind Einträge, die schnell gefunden werden müssen im vorderen Bereich des OD?
- Korrelieren *ObjectType*, *Type* und *Entry.Data.Data* bzw. *Entry.Data.DataPtr* miteinander?
- Sind die Zugriffsrechte im Element *Attribut* richtig gesetzt?
- Stimmt die Datenbreite im OD mit der tatsächlich definierten überein?
- Wurden alle Array's und Strings richtig initialisiert? Dies geschieht üblicherweise im Callback Initialization?
- Werden die Elemente von Strukturen, die als Einheit angesprochen werden, im Speicher wirklich direkt hintereinander abgelegt?

4.10 Nachrichtenverarbeitung

Die Nachrichterverarbeitung in der CANopen-Software erfordert gesonderte Betrachtungen, da sie die unterschiedliche Behandlung von Ereignissen auf PC's und eingebetteten Systemen vereinen muß. Software auf handelsüblichen PC's ist ereignisgesteuert. Ein Programm befindet sich üblicherweise in einer Warteschlange und wird erst aktiviert, wenn ein Ereignis stattfindet. Bei eingebetteten Systemen wird meistens das Polling-Verfahren angewendet, d.h. ein Programm befindet sich in einem Loop und testet regelmäßig, ob ein Ereignis stattgefunden hat.

[10] Normalerweise ist folgende Schreibweise vorzuziehen: BINT((ID_EDGES<<16) | 56). Doch der Keil-Compiler scheint diese Syntax in Preprozessoranweisungen nicht korrekt zu übersetzen. Stattdessen ermittelt er als Ergebnis 56.

In der CANopen Software werden drei verschiedene Modi der Nachrichtenverarbeitung unterschieden:

- Abarbeitung im Interrupt
- Polling-Loop
- Ereignisgesteuerte Abarbeitung

Die ***Abarbeitung von Nachrichten im Interrupt*** stellt die mit Abstand schlechteste, aber einfachste Lösung dar. Bei Eintreffen von Nachrichten werden diese sofort weiterverarbeitet und gegebenenfalls beantwortet. Der laufende Task wird für die komplette Bearbeitungszeit unterbrochen. Diese Art der Nachrichtenverarbeitung ist weiterhin gekennzeichnet durch:

- Lange Interruptabarbeitungszeiten, wodurch Überlaufgefahr der internen CAN-Controller MessageQueue besteht
- Probleme der Datenkonsistenz im Falle eines Zugriffs auf Speicherzellen, die von Interrupt und Haupttask gleichzeitig benutzt werden.
- Einbindung in Multitaskingsysteme schwierig bzw. unsauber, da alle anderen Tasks blockiert sind

Der ***Polling-Loop*** wird vorwiegend in eingebetteten Systemen verwendet. Die prinzipielle Arbeitsweise ist folgende: Die Hardware löst beim Empfangen einer Nachricht einen Interrupt aus. Dieser bearbeitet die empfangenen Daten und sortiert die Nachricht in eine Nachrichtenwarteschlange (MessageQueue) ein. Danach wird der Interrupt verlassen. Das Hauptprogramm wird wieder aktiviert und kann durch Auslesen der Warteschlange ermitteln, ob eine Nachricht empfangen wurde und diese dann auswerten. Diese Variante hat entscheidende Vorteile gegenüber der Abarbeitung im Interrupt:

- Der Interrupt wird schnell verlassen. Die CAN-Controller werden durch das schnelle Rücksetzen der Hardware sofort wieder bereit, um neue Nachrichten zu empfangen.
- Es treten kein Konflikte hinsichtlich der Datenkonsistenz auf, da auf alle Daten ausser auf die Messagequeue lediglich im Haupt-Kontext (Haupttask) zugegriffen wird.

Der Nachteil des Polling-Loops besteht in dem möglichen Verlängern von Antwortzeiten und in der ständigen Belastung der CPU, was in eingebetteten Systemen selten zum Problem wird, da es meistens nur einen Task gibt. Nur könnte es sich nachteilig auf Stromverbrauch und Sleep-Modi auswirken.

In der ***ereignisgesteuerten Abarbeitung*** werden ähnliche Prinzipien wie beim Polling-Loop verwendet. Der Unterschied besteht darin, dass nach dem Einsortieren der Nachrichten in die MessageQueue ein betriebssystemspezifisches Ereignis erzeugt wird. Nach der Auslösung des Ereignisses und Beenden des Interrupts wird die Wartefunktion des Threads beendet und kehrt zum Hauptprogramm zurück, um die Nachricht aus der Warteschlange auszulesen und auszuwerten. Zu beachten ist, dass die Routine, die die Nachrichten einsortiert u. U. gleichpriorisiert zu der Abarbeitungsroutine ist. Dies hat Konsequenzen auf die Funktionen des kritischen Abschnitts und erfordert Anpassungen der CAN-Treiber[1]. Zu Eigenschaften der ereignisorientierten Abarbeitung zählen:

- Verhinderung der Blockierung von anderen Tasks & Threads sowie Veringerrung der CPU-Last durch Wartefunktionen.
- Undefinierte Antwortzeiten für Nachrichten in Multitaskingsystemen – es sei denn, der Task erhält die höchste Priorität.

4.11 Remote Control

Die Einführung des Moduls *CO_RemoteControl.c* versetzt die CANopen Software in die Lage, von außen ferngewartet zu werden. Es handelt sich im Prinzip um eine zweite Kommunikationseinrichtung zum Gerät. Im Normalfall ist dies eine zusätzliche serielle Verbindung (RS232). Es ist allerdings auch denkbar, die Fernwartung über eine Tunnelung von SDO's durchzuführen.
Bei der Kommunikation mit dem Zielgerät werden einfache ASCII-Streams übertragen. Um die Abarbeitung zu beschleunigen, wurden die Befehle in Zahlen kodiert und als ASCII-Zahlen übertragen. Die Streams werden in dem Gerät mittels getc() von *stdin* gelesen. Alle Ausgaben werden nach *stdout* geschrieben. Der Vorteil dieser Variante besteht darin, dass die Ein- und Ausgabenroutinen standardisiert sind und von jedem ANSI-C Compiler unterstützt werden, wodurch auf sehr einfache Weise eine Fernwartung möglich wird. Ein weiterer Vorteil bei der Verwendung von Streams ist der einfacher Aufbau und das leichte Interpretieren des Inhalts von übertragenen Streams.
In diesem Zusammenhang muß beachtet werden, dass die Verbindung zwischen Leit-PC und Remote-PC nicht echtzeitfähig sein kann, was aber für eine erfolgreiche Fehlerkorrektur nicht entscheident ist. In so einem Fall ist es wichtig, das Netzwerk so zu konfigurieren, dass der Fehler behoben wird. Prinzipiell sind aktuelle nicht-echtzeitfähige Netzwerke, wie Ethernet-LAN's bzw. Standleitungen schnell genug, um Daten in einem erforderlichen Zeitfenster sicher zu übertragen, eben nur nicht deterministisch.

Auf eine Typüberüfung der übergebenen Parameter in den Remote-PC's zur Laufzeit wurde verzichtet, um das Module möglichst klein und effizient zu halten. Das Zielgerät kann nur aufgrund des Stream-Formats auf die dazugehörigen Typen schließen.
Das Setzen eines Eintrages im Object Dictionary besteht z.B. aus folgenden Zahlen: „1 2 3 5". Die 1 spezifiziert den ***remote control***, den Zahlencode für das Kommando. Die nachfolgenden Parameter sind alle befehlsabhängig. In diesem Fall legen die Zahlen 2 und 3 den Multiplexor [2][3] fest. Die 5 ist der Wert, der in das Objekt [2][3] geschrieben werden soll. Es gibt aber keine Möglichkeit festzustellen, ob 5 ein gültiger Wert für dieses Objekt ist. Dies ist dem Anwender der Fernwartungskonsole überlassen. Er muss sich vorher eingehend mit den Grenzen der Befehle und der Objekte auseinander setzen. In dem Beispielstream könnte statt der 5 auch ein String oder eine Kette von Strings übergeben werden. Diese würden dann byteweise in das angegebene Objekt geschrieben werden.
Eine ausführliche Übersicht der Fernwartungskommandos ist in Anhang 7B gezeigt. Eine genaue Kenntnis der Befehlsnummern ist im Grunde genommen nicht notwendig, da Steuerungsprogramme, wie die *CANopenConsole*, Synonyme in Textform verwenden, um dem Anwender die Arbeit zu erleichtern (s. Kap. 5.4)

4.12 Entwicklungsumgebung

4.12.1 Compiler

Die CANopen-Software wurde mit den Compilern Visual C++ V6.0 von Microsoft und dem Keil C-Compiler V4.10 (µVision V2.10) zeitgleich entwickelt. Eines der eingangs erwähnten Ziele war, die Software komplett plattformunabhängig in ANSI-C zu entwickeln. Dadurch bestand die Möglichkeit, den größen Teil mit dem MS Visual C++ zu realisieren. Dieser Compiler hatte deutliche Vorteile gegenüber dem Keil, z.B. einen brauchbaren und funktionstüchtigen

Debugger[11] und der Verzicht auf einen Dongle[12]. Ein besonderes Gimmnick ist die Möglichkeit des Visual C++, während des Debuggends Programmcodeänderungen durchzuführen, so war es möglich, die komplette CANopen Software auf einem Windows-Rechner zu programmieren und zu simulieren. Downloads der Software auf die Zielgeräte waren selten nötig. Sie sollten allerdings in regelmäßigen Abständen durchgeführt werden, da sich häufig Fehler einschleichen, wie z.B. die Definition von vorinitialisierten Array's, die dann plötzlich nicht gefüllt sind.

Bei der Entwicklung wurde weiterhin darauf geachtet, die Arbeit für den späteren Anwendungsentwickler möglichst einfach zu gestalten. Bei der Entwicklung eigener Anwendungen muss dieser z.B. nur die Datei *include/CANopen/CANopen.h* in das Projekt einbinden, um eine CANopen-Unterstützung mitzukompilieren. Diese Headerdatei lädt alle weiteren benötigten Dateien automatisch hinzu.
Das Linken erfordert etwas mehr Initiative seitens des Anwendungsentwicklers, da es sehr compiler-spezifisch ist. Eine Möglichkeit besteht darin, alle Quellcodedateien des Verzeichnisses *include/CANopen/*.c* in das Projekt miteinzubinden, mitzukompilieren und dann automatisch mitzulinken. Man spricht von *statischem Linken*. Erstellt man aus allen CANopen-Dateien eine Bibliothek (DLL unter Windows) und bindet sie über die *CANopen.lib* dynamisch dem Projekt hinzu, handelt es sich um *dynamisches Linken*. Der Vorteil besteht darin, dass man Kunden im Bedarfsfall eine neuere, korrigierte CANopen-Library zuschicken kann, ohne das diese ihr Projekt ändern bzw. neu kompilieren müssen. Auf diese Weise kann verhindert werden, dass der Quellcode der CANopen-Implementierung an Kunden weitergegeben werden muss.

4.12.2 Verzeichnishierachie

CANopen ist ein sehr komplexes Übertragungsprotokoll, das eine Menge an sog. *Services* unterstützt, die z.T. optional sind. Die Komplexität erfordert es, den Quellcode zu strukturieren und in Module aufzuspalten. Dadurch wird die Wartung und das spätere Hinzufügen weiterer Services deutlich vereinfacht. Die derzeitige Implementierung des CANopen-Protokolls teilt sich wie in Tabelle 11 dargestellt auf. Jedes der gezeigten Module besteht aus einer Header- und einer Quellcodedatei, in der die jeweiligen Funktionen definiert sind.

Modul	Aufgaben
CANopen.h	Diese Headerdatei ist die einzige Datei, die von den Anwendungen eingebunden werden muss. Alle anderen werden bei Bedarf automatisch hinzugeladen. • Einbinden aller nötigen Dateien • Deklaration allg. Datenstrukturen, wie NODE-ID, MUX, CO_HANDLE • Nachrichtenauswertung und Verteilung in die einzelnen Module • Auf- und Abbau von CANopen Instanzen

[11] Allerdings wird bei dem Einsatz verschiedene Compiler deutlich, dass das Verhalten trotz ANSI-C nicht immer gleich ist, zu sehen am Beispiel:

```
#ifdef __KEIL__
#define BPTR(pointer)     { (const UINT32 *) &(pointer) }
#else
#define BPTR(pointer)     { (const UINT32 ) &(pointer) }
#endif
```

[12] Ein Dongle zeigt seine wahren Vorzüge erst bei der Arbeit mit mehreren Rechnern.

Modul	Aufgaben
CANopenLib.h	• Deklaration der Funktionsnamen, die der Anwendung zur Verfügung gestellt werden
CANopen_NMT.h	• Network Management • Senden von NMT-Requests
CANopen_NMTEC.h	Network Management Error Control, beinhaltet abhängig von den Compiler-Schaltern: • Lifeguarding • Heartbeat
CANopen_ObjectDictionary.h	• Behandlung und Verwaltung des Object Dictionaries • Lesen und Schreiben von Einträgen
CANopen_PDO.h	• Auswerten und Initiieren von PDO-Nachrichten
CANopen_RemoteControl.h	• Unterstützung für Fernwartung durch einfache ASCII-Befehle
CANopen_SDO.h	• Allg. Deklarationen zur Verwendung von SDO's
CANopen_SDOClient.c	• Modul zur Unterstützung der SDO-Client Funktionalität (Starten von SDO-Anfragen)
CANopen_SDOServer.c	• Auswerten von empfangenen SDO-Anfragen aus dem Netzwerk
CANopen_SYNC.h	• Senden und Empfangen von Sync-Objekten

Tabelle 11 Module der CANopen Software

4.12.3 Frameworks

Dem Entwickler soll in der folgenden Darstellung (Tabelle 12) eine Übersicht gegeben werden, wie er möglichst einfach seine CANopen Anwendung strukturieren kann. Man kann vereinfacht von einem *Framework* sprechen, welches dem Entwickler erlaubt, auf sehr schnelle Art und Weise das CANopen-Netzwerk ohne Kenntnis der inneren Strukturen der CANopen-API in eigene Applikationen einzubinden.

Modul	Aufgaben
COEnvironment.h	Definition aller gerätespezifischen Parameter. Dieses Modul wird in jedem Gerät für jede CANopen Instanz einmal angelegt (Geräteanwendung). • Auswählen des CAN-Controllers • Auswerten von CO_Callbacks • Konfiguration der CANopen Umgebung • Anlegen des Object Dictionaries • Behandlung von entfernten Objekten • Binbindung in das Betriebssystem
COApplication.h	Definition aller anwendungsspezifischen Parameter - wird einmal für die komplette CANopen Anwendung angelegt und in jede Geräteanwendung eingebunden. • Deklaration der ID's aller Object Dictionaries • Festlegen der Node-ID’s

Modul	Aufgaben
CANEnvironment.h	Definition der CAN-Controller spezifischen Parameter. Das Modul wird für jeden CAN-Controller angelegt. • Initialisierung der CAN-Hardware • Definition des Interrupteinsprungs für das Signalisieren bei Eintreffen neuer Nachrichten
COApplicationDebug.h	Definition von anwendungsspezifischen Funktionen, die im Debug-Modus zusätzlich benutzt werden. Dieses Modul wird für die gesamte CANopen-Anwendung einmal angelegt und in jede Geräteanwendung eingebunden. • Ausgabe von Informationen bei Aufruf der CO_Callbacks
Main.c	Synonym für die Initialisierung innerhalb der Geräteanwendung. • Auswahl des COEnvironments • Anlegen des CANopen-Handles • Initialisierung der CANopen Instanz durch Aufruf der Funktion CO_InitCANopen() • Einsprung in die Nachrichtenverarbeitung • Zerstörung der CANopen Instanz durch Aufruf der Funktion CO_EndCANopen()

Tabelle 12 Vorschlag für die Strukturierung einer CANopen-Anwendung

Die Main-Funktion in einer Geräteanwendung gestaltet sich nahezu trivial, wie in Abbildung 4-17 gezeigt wird. Die einzige Größe, die prinzipiell variabel ist, ist die Bezeichnung des Callbacks zur Initialisierung der CANopen Instanz.

```
#include <stdio.h>
#include <CANopen/CANopen.h>

OPTOERROR APP_ConfigureCANopen(CO_HANDLE *cohandle);        // external, defined in COEnvironment.c

void main()
{
        CO_HANDLE       handle;                             // CANopen Handle
        char            buffer[CONSOLEBUFFERSIZE];          // buffer for remote commands

        if ( ! (CO_InitCANopen(&handle, APP_ConfigureCANopen)) ) // init CANopen
        {
                TRACE1("Camera: %d ready!\n",(int)(handle.NodeID));
                while (1)                                   // endless loop
                {
                        CO_MessageLoop(&handle);            // call message loop
                        if ( GetRemoteCommand(buffer) )     // new remote command received ?
                                CO_RemoteCommand(&handle, buffer);
                }
        }
        CO_EndCANopen(&handle);
}
```

Abbildung 4-17 Beispiel einer CANopen Initialisierung

Die dargestellte Main-Funktion ist ein Auszug aus einer Geräteanwendung, die in einem eingebetteten System mit C166-Mikrocontroller und SJA1000 CAN-Controller lauffähig ist. Sie

könnte auch auf anderen Plattformen verwendet werden, da in ihr kein einziger direkter Zugriff auf die Hardware erfolgt. Die wichtigsten Schritte im Einzelnen sind:

- Einbinden der CANopen-API und Includes von ANSI-C
- Deklarieren der Callback-Funktion APP_ConfigureCANopen, die im Modul COEnvironment definiert ist. Die Verbindung zwischen Main-Funktion und APP_ConfigureCANopen wird beim Linken hergestellt.
- Anlegen der CANopen-Instanz (COHandle)
- Aufruf der CO_InitCANopen() Funktion, um das COHandle und den CAN-Controller zu initialisieren.
- Ausgabe, das Initialisierung erfolgreich war
- Eintritt in Endlosschleife
- Aufruf der Funktion zur Nachrichtenverarbeitung, die alle Nachrichten in der Warteschlange abarbeitet.
- Lesen eines Kommandos der Fernwartung und Ausführung der entsprechenden internen Funktion

4.12.4 Redirection von printf()

In dem Modul CO_RemoteControl wird statt der üblichen printf() Funktion ein Macro _printf() zum Ausgeben der Daten benutzt. Dies ist nötig, da in Konsolenanwendungen unter Windows/Linux ein *printf* sofort ausgegeben wird. Der Anwender möchte aber möglicherweise wissen, von welchem Gerät die Ausgaben sind, die auf dem Bildschirm stehen, sodass dem auszugebenden Text ein Präfix vorangestellt wird, z.B. *DEV1:* für Gerät 1. Zu diesem Zweck müssen die printf() Anweisungen umgeleitet werden, wenn die CANopen Software auf dem selben Rechner wie das Administrierungsprogramm läuft.

In Geräten, die über RS232 kommunizieren, wird die Ausgabe von *printf* über die serielle Schnittstelle transportiert und im Fernwartungsprogramm ausgelesen. Hier ist nun das Voranstellen der Gerätenummer kein Problem. Das unterschiedliche Verhalten des Macros wird bestimmt durch den Compilerswitch *_REDIRECT_PRINTF* (s. Kap. 4.12.5). Das Problem trifft alle Ausgaben, sodass auch TRACE-Kommandos intern mit dem _printf() Macro arbeiten.

4.12.5 Compiler-Switches

Zur Optimierung der CANopen-Software stehen eine Reihe von Compiler-Switches zur Verfügung. Der Speicherverbrauch und die CPU-Belastung kann durch geeignete Maßnahmen deutlich (bis zu 50% gesenkt werden). Der Hintergrund ist, dass viele Geräte gewisse Dienste des CANopen Protokolls nicht unterstützen müssen und auf diese Weise abgeschaltet werden können. Die unterstützten Compiler-Schalter sind im Anhang (A Compiler-Switches für die CANopen Software) dargestellt.

4.13 Einschränkungen

Die entwickelte CANopen-API hat einige Einschränkungen, die dem Entwickler bekannt sein müssen. Die Einschränkungen betreffen in der Regel nur Komponenten, deren Implementierung als optional gekennzeichnet ist:

Einschränkung	Beschreibung
Verzicht auf CRC-Überprüfung	Die gesendeten Daten von SDO's können durch CRC-Checksummen kontrolliert werden. Die Implementierung ist optional.
Fehlende Unterstützung von SDO-Block Diensten	Die in der CANopen Spezifikation definierten SDO-Dienste beinhalten neben dem SDO Download/Upload auch einen SDO Block Download/Upload. Dieser Service wird vorwiegend bei sehr langen Datenübertragungen benutzt und zeichnet sich durch eine Nummerierung der Blöcke aus, die einzeln CRC-geschützt sind. Auf diese Weise wird eine einfachere Fehlererkennung und -korrektur unter Zuhilfenahme von Go-Back-N Mechanismen möglich. Die Implementierung ist optional.
Keine Unterstützung von Emergency Objects	Im Fehlerfall wird eine Benachrichtigung über die Fehlerquelle an die entsprechende Consumer im Netz verschickt. Die Implementierung ist optional.
Pre-Defined Connection set	Die CANopen-API ist derzeit nur in der Lage, das *pre-defined connection set* zu verwalten, das aber in den meisten Anwendungsfällen ausreicht.

Tabelle 13 Einschränkungen in der CANopen-API

In der CAN-API gibt es derzeit die Einschränkungen, dass alle CAN-Controller Treiber nur eine CAN-Instanz öffnen können.
Im Falle des Treibers für die PCI-Karte von IXXAT liegt das Problem an der benutzten VCI-Bibliothek, die sich nur einmal öffnen läßt.
Der SJA1000 CAN-Controller kann nur einmal instanziiert werden, da im C166 Programm keine *malloc()* Funktion zur Verfügung steht und demnach kein CAN-Handle dynamisch erzeugt werden kann. Es wird daher statisch angelegt und kann nur einmal benutzt werden. Prinzipiell kann der Treiber erweitert werden und ohne Probleme mit mehreren Instanzen arbeiten.

5 Die CANopen Console

5.1 Einleitung

Die ***CANopen Console*** ist ein Administrierungstool zur Überwachung eines CANopen Netzwerkes. Es handelt sich um ein Terminalprogramm, indem Kommandos von *stdin* gelesen werden und Ausgaben nach *stdout* geschrieben werden. Es gibt keine Fenster oder Knöpfe. Der Vorteil einer Konsolenanwendung besteht in der einfachen Portierbarkeit der Software und darin, das zu steuernde Netzwerk äußerst flexibel und ohne großen Aufwand umzukonfigurieren bzw. zu erweitern.
Die herausragende Eigenschaft des vorgestellten Programms besteht darin, die CANopen Geräte fernwarten zu können. Um die vollständige Funktionalität des Konsolenprogramms zu gewährleisten, müssen alle Geräte des Netzwerkes das Modul *CO_RemoteControl* enthalten, anderenfalls ist das Gerät für die CANopen Console nicht erreichbar.

Die CANopen Console gliedert sich in mehrere Funktionsblöcke, die in Abbildung 5-1 dargestellt sind. Die Eingabe der Befehle (s. Kap. 5.4) erfolgt in dem Block *Console*. Dieses Modul repräsentiert das Terminalprogramm. Es ist die Verbindungsstelle zum Betriebssystem. Die nachfolgenden Funktionsmodule sind unabhängig von dem Eingabeteil, sodass es jederzeit möglich ist, anstatt der Eingabeconsole eine angepasste MFC bzw. QT-Applikation zu verwenden.

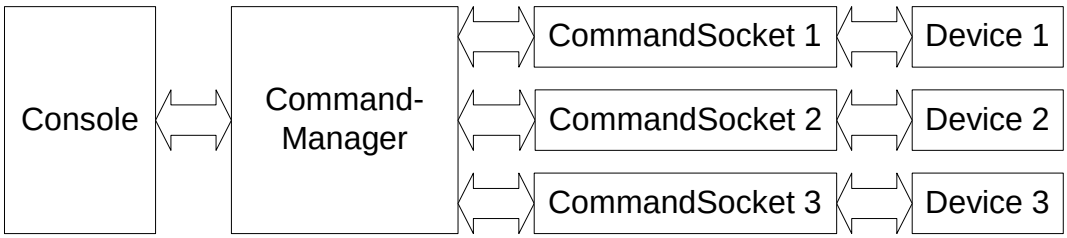

Abbildung 5-1 Gliederung der CANopen Console

Das Modul ***CommandManager*** kennzeichnet eine Library, die die Aufgabe hat, die eingegebenen Befehle an die entsprechenden CANopen-Geräte weiterzuleiten, alle Ausgaben der CANopen Geräte nach *stdout* mit einem Prefix wie *DEVx:* zu versehen und in der Konsole auszugeben. Die Geräte können dabei auf verschiedenste Art und Weise an den PC angekoppelt sein, z.B. COM1, COM2 bzw. LPT1. Die Module ***CommandSockets***[13] stehen für das Kommunikationsmodul zwischen dem CommandManager und den Geräten. Jedes CommandSocket wird als Bibliothek abgelegt, ist aber abhängig vom CommandManager.
Alle Verbindungen sind bidirektional. Ein- und Ausgaben werden in ASCII-Streams zwischen den Modulen ausgetauscht.

[13] Socket[engl], Steckdose. In der Fachsprache spricht man von Kommunikationsendpunkten.

5.2 Der CommandManager

Der CommandManager unterteilt sich in weitere Funktionsmodule (s. Abbildung 5-2), die kurz erklärt werden. Er wurde komplett in C++ entwickelt und wird als eigenständige Bibliothek (z.B. einer Windows-DLL) erstellt. Diese Library ist in der Lage, intern definierte Klassen zu exportieren, sodass sie in anderen Libraries, z.B. den CommandSockets, als Basisklasse verwendet werden können.

Die ***CANopenSession*** ist das zentrale Element des CommandManagers. Durch das Design dieses Moduls ist es möglich, mehrere CANopen Sitzungen zu verwalten, d.h. es kann gleichzeitig mehr als ein CANopen-Netzwerk verwaltet werden. Das setzt allerdings voraus, dass auch alle anderen Komponenten in der Lage sind, mehrere Sessions zu verwalten.

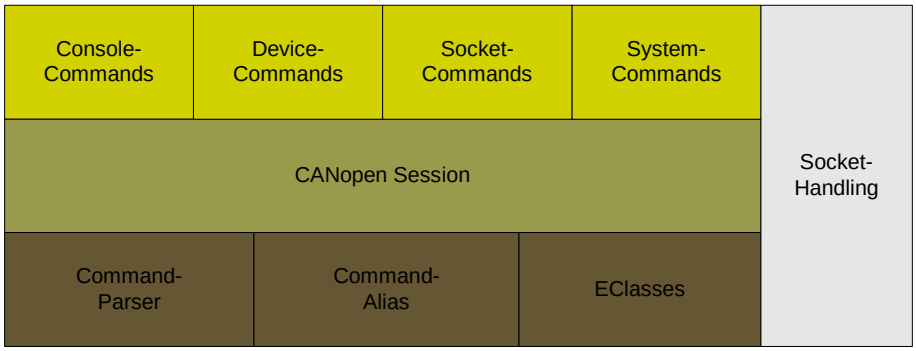

Abbildung 5-2 Gliederung des CommandManagers

Die Eingabe von Befehlen erfolgt im Modul *Console*. Sie hat dabei keinerlei Kenntnis über Syntax und Typ der Befehle. Sie liest die Befehle ein und gibt diese sofort an den dynamisch gelinkten CommandManager weiter. Dadurch ergeben sich die in Kap. 4.11 erläuterten Typprobleme.
Die Interpretation der Befehle wird von einem eigenen ***CommandParser*** übernommen. Er zerlegt die Befehle in ihre Bestandteile, prüft deren Syntax und leitet sie an eines der vier Kommandoauswertungsmodule weiter:

- ConsoleCommands — Es handelt sich um Kommandos, die die Bedienung der Konsole beeinflussen und unabhängig von CANopen Geräten sind.
- SocketCommands — Kommandos, die das Verhalten der Sockets zu den entsprechende CANopen-Geräten beeinflussen, z.B. Einstellen der Baudrate.
- DeviceCommands — Kommandos, die das Verhalten der CANopen-Geräte direkt beeinflussen, z.B. durch Ändern eines Eintrages im Object Dictionary.
- SystemCommands — Kommandos des Betriebssystems

Die Kommunikation zwischen einem CommandSocket und einem CANopen Gerät wird in einem eigenen Threads abgewickelt. Dies ist notwendig, um gegenseitige Blockierungen bei dem Verwalten mehrerer CANopen Geräte zu vermeiden und schnell auf den Empfang von Nachrichten reagieren zu können. Die Verwendung von Threads verbessert ausserdem das Laufzeitverhalten auf Maschinen im SMP-Betrieb.
Bei der Benutzung von Threads ist es sehr wichtig, diese miteinander zu synchronisieren und gleichzeitige Zugriffe auf Variablen zu verhindern. Die ANSI-C Bibliothek stellt dafür keine

Funktionen zur Verfügung, sodass diese Aufgaben stark betriebssystemabhängig und problematisch in Bezug auf das Ziel sind, die Software plattformunabhängig zu gestalten. Aus diesem Grund wurden alle Aufgaben, die Threads, Wartefunktionen und Signalbehandlungen beinhalten, in eigene Klassen verlegt. Diese Klassen sind im Modul ***EClasses*** definiert. Diese kleine Klassenbibliothek wird beim Laden des CommandManagers dynamisch hinzugelinkt. Es ist eine eigenständige Bibliothek, die auch in anderen Projekten benutzt werden kann.
Die einzelnen Teilklassen sind in Tabelle 14 gezeigt. Alle Module des CommandManagers sind so programmiert, dass sie plattformunabhängig sind und bei kritischen Aufgaben, wie die Arbeit mit Threads, auf die Bibliothek EClasses zurückgreifen.
Für den Fall des Plattformwechsels müssen lediglich die Klassen *EThread* und *EEvent* dieser kleinen Klassenbibliothek angepasst werden. Zurzeit existiert nur ein Windows-Version dieser beiden Klassen.

Klassen	Beschreibung
EString	Verwalten von dynamischen Strings auf einfachste Weise – plattformunabhängig
EFile	Komfortables Laden von Dateien – plattformunabhängig
EThread	Starten von Threads
EEvent	Events zum Synchronisieren von Threads
EHandshake	Kombination von Events zum Datenaustausch – plattformunabhängig, basiert auf EEvent

Tabelle 14 Klassen des Moduls EClasses

Das Modul ***SocketHandling*** beinhaltet die Basisfunktionalitäten für alle CommandSockets und die interne Verwaltung der Sockets. Mit Hilfe dieses Moduls ist es möglich, die Socket-Bibiotheken zu laden, Threads zu starten, zu synchronisieren und zu beenden sowie die Konfiguration der Sockets durchzuführen, z.B. Setzen der Baudrate. Teile dieses Softwaremoduls werden exportiert und den Socket-Bibliothen zur Verfügung gestellt, z.B. die Basisklasse *CommandSocket*.

5.3 CommandSocket Bibliotheken

Ein Administrierungsprogramm wie die CANopenConsole muss eine Möglichkeit besitzen, das Verhalten der Netzwerkteilnehmer zu beeinflussen. Auf Seiten der CANopen-Geräte wurde dazu das Modul *CO_RemoteControl* entwickelt. Durch dieses Modul kann das Gerät über die Kanäle *stdin* und *stdout* ferngesteuert werden, die z.B. an die serielle Schnittstelle gebunden sind. Falls ein CANopen Gerät nicht über die serielle, sondern über die parallele Schnittstelle angeschlossen ist bzw. sich im gleichen Rechner befindet (PCI-Karte), versagt diese Lösung. Aus diesem Grund muss zwischen den verschiedenen Kommunikationsmethoden zur Fernwartung unterschieden werden. Dies wird erreicht, indem die Kommunikation mit jedem Gerät über eine Kommunikationsbibliothek durchgeführt wird, dem ***CommandSocket***. Bei der Initialisierung des Gerätes wird dem Kommando *InitSocket* der Name der Bibliothek übergeben.
Für den CommandManager existieren derzeit die Bibliotheken CommandSocketRS232 und CommandSocketVCIPCI. Es ist darüber hinaus geplant, die Kommunikation über eine Tunnelung von CANopen Objekten durchzuführen, sodass auf eine zusätzliche physikalische Verbindung verzichtet werden kann.

5.3.1 CommandSocketRS232

Die Kommunikation mit dem CANopen Gerät erfolgt über die serielle Schnittstelle bei einer Standardbaudrate von 9600 Baud. Sie ist prinzipiell konfigurierbar. Eine Übersicht über den Informationsfluß zeigt Abbildung 5-3.

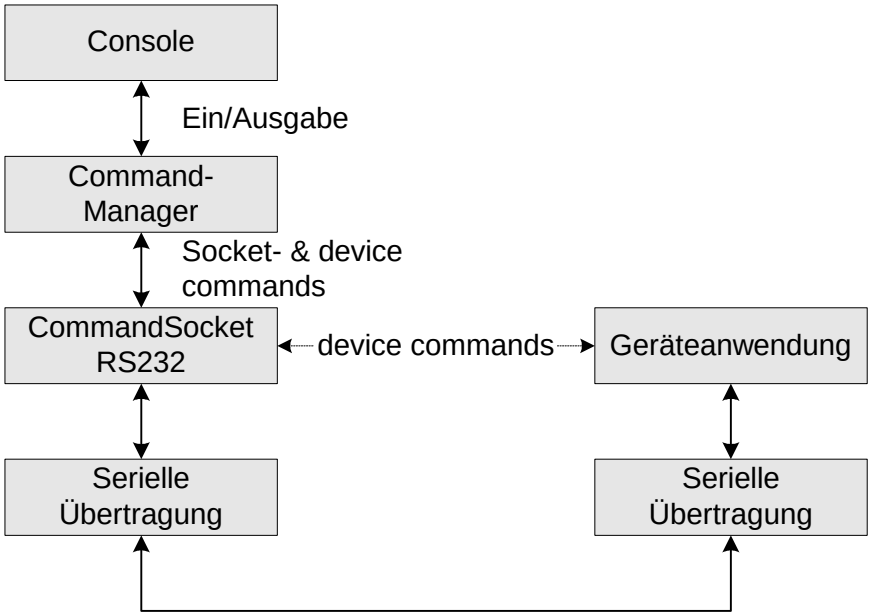

Abbildung 5-3 Informationsfluß bei serieller Anbindung von CANopen Geräten

Bei Benutzung der Socketkommandos ***InitSocket*** und ***ConfigSocket*** besteht die Möglichkeit, Parameter zu übergeben:

- Port — Festlegen des Kommunikationsports, z.B. COM1 oder COM2. Der Portname muss bei der ersten Initialisierung angegeben werden.
- Hexfile — Name der Downloaddatei setzen. Diese Datei wird bei Bedarf gesucht und zum Gerät heruntergeladen. Dieser Parameter ist für die Initialisierung nicht zwingend erforderlich.
- Baudrate — Setzen der Baudrate für die serielle Kommunikation zwischen PC und CANopen Gerät. Standardmäßig werden 9600 Baud benutzt.

Diese Library besitzt die Möglichkeit, Downloads auf einzelne oder gleichzeitig auf alle Geräte durchzuführen. Dies ist besonders praktisch, wenn in der CANopen Anwendung ein Fehler behoben wurde und per Skript alle Geräteanwendungen neu kompiliert wurden. Nach der Erstellung können die verschiedenen Mikrocontrollerprogramme durch Aufruf eines Befehls, wie z.B. „socket 0 download“, auf die entsprechenden Geräte geladen werden. Die Gerätenummer null zeigt einen Broadcast an, sodass dieses Kommando an alle CommandSocket-Bibliotheken weitergeleitet wird, in denen dann automatisch das bei InitSocket angegebene Programm(Hexfile) für den Download benutzt wird.

Der Nachteil der Anbindung von CANopen-Geräten durch diese Bibliothek besteht in der zusätzlichen physikalischen Verbindung zwischen PC und Gerät. Vor allem bei langen Entfernungen ist diese Lösung unbrauchbar.

5.3.2 CommandSocketVCIPCI

Diese Bibliothek dient zum Datenaustausch zwischen der PC-Implementierung einer CANopen Geräteanwendung und dem CommandManager. Eine Übersicht über den Informationsfluß zeigt Abbildung 5-4.

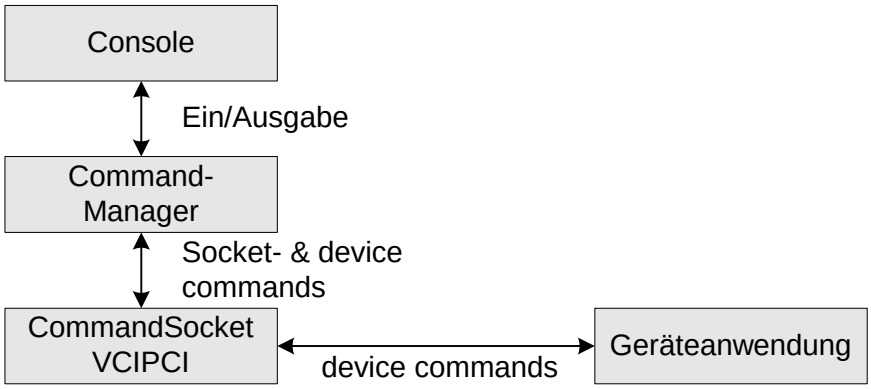

Abbildung 5-4 Informationsfluß bei direkter Speicheranbindung

In dieser Library ist der Compiler-Schalter REDIRECT_PRINTF von Bedeutung, um die Ausgaben der CANopen-API umzuleiten und einen entsprechenden Präfix hinzuzufügen. Zur korrekten Initialisierung erwartet die Bibliothek als einzigen Parameter den Namen der zu ladenen Geräteanwendung, die ebenfalls als DLL angelegt ist. Eine Initialisierungssequenz könnte folgendermaßen aussehen: *initsocket 1 CommandVCIPCI.dll library=Camera1.dll*

Bei einigen Applikationen kann es von Vorteil sein, auf den CommandManager und damit auch auf die CommandSockets zu verzichten und direkt mit der Geräteanwendung zu kommunizieren. Durch die Trennung von CommandSocketVCIPCI und Geräteanwendung wird dem Entwickler diese Möglichkeit gegeben. In Abbildung 5-5 ist ein solches Szenario dargestellt. Der direkte Zugriff entspricht annähernd dem Verhalten auf eingebetteten Systemen.
Die Schnittstelle für den direkten Zugriff ist nicht weiter definiert und wird dem Entwickler überlassen.

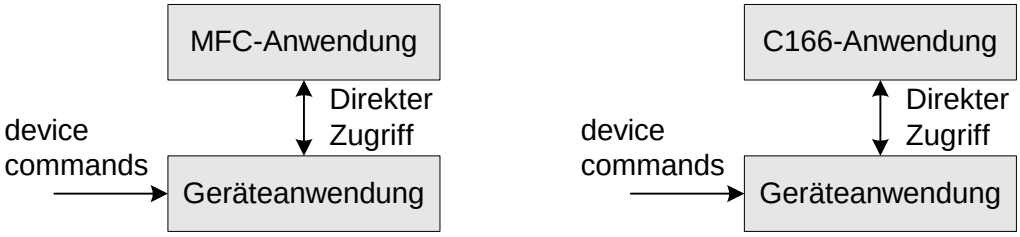

Abbildung 5-5 Direkter Zugriff auf CANopen Geräteanwendungen

Der Entwickler sollte jedoch in seinen Applikationen darauf verzichten, die Geräteanwendungen komplett in das Projekt hineinzulinken, denn durch Verwendung einer Library wird erreicht, dass die CANopen-internen Komponenten nur über standardisierte Schnittstellen angesprochen werden.
In Abbildung 5-6 ist eine MFC-Anwendung gezeigt, die eine CANopen-Geräteanwendung beinhaltet und ein komplettes Netzwerk warten kann. Die Geräteanwendung könnte z.B. die Auswertung von Messdaten übernehmen (Leit-PC). Bei Schwierigkeiten im Netzwerk kann zusätzlich ein Terminalfenster zu Debug-Zwecken geöffnet werden und so das Problem durch geeignete Massnahmen eingegrenzt werden. Ist der Fehler lokalisiert, kann im Anschluss z.B. ein Softwareupdate für alle Netzwerkgeräte durchgeführt werden.

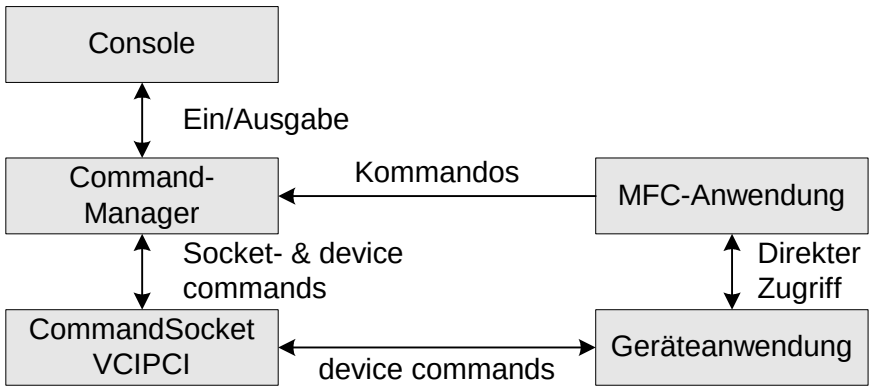

Abbildung 5-6 PC-CANopen Geräteanwendung inkl. Fernwartung des Netzwerkes

5.4 Bedienung der Console

5.4.1 Console Commands

Der CommandManager unterstützt eine Vielzahl von internen Kommandos (*Consolecommands* bzw. *Konsolenkommandos*). Hervorstechend ist dabei vor allem die Eigenschaft des ***Scriptings***, d.h. des Ausführens eigener ASCII-Skripte, die geladen und ausgeführt werden, um ein CANopen Netzwerk zu booten bzw. zu konfigurieren. Jede Zeile in einem Skript entspricht einem Befehl der Eingabekonsole. Durch die Verwendung eines eigenen *CommandParsers* ist es möglich, unter verschiedenen Betriebssystemen dieselben Skripte zu verwenden.[14]
Desweiteren gibt es die Möglichkeit, über das interne Kommando *alias* Kurzformen bzw. Synonyme für ganze Befehle zu erstellen. Diese Funktionalität wird durch das Modul ***CommandAlias*** (s. Abbildung 5-2) bereitgestellt.

5.4.2 Socket Commands

Die SocketCommands dienen ausschließlich zum Verwalten und Konfigurieren der Sockets. Zurzeit gibt es drei Befehle dieser Art: *initsocket*, *configsocket* und *socket*. Alle drei Befehle

[14] Unter Windows werden in ASCII-Dateien die Zeilenenden üblicherweise mit 0x0d, 0x0a abgeschlossen. Unter Unix-System lediglich mit 0x0a.

erwarten als ersten Parameter die Gerätenummer. Sollte eine Null angegeben werden, betrifft das Kommando alle Geräte (Broadcast).

- ***Initsocket*** wird benutzt, um eine Kommunikation mit einem CANopen Gerät zu initialisieren. Dazu wird dem Befehl die Gerätenummer und der Name einer Bibliothek übergeben. Diese Bibliothek stellt die Verbindung zu dem CANopen Gerät her und kommuniziert mit diesem über Befehle, die in Kap. B-Fernwartungskommandos beschrieben sind. Alle weiteren Parameter für Initsocket sind Socket-spezifisch.
- ***Configsocket*** dient zum nachträglichen Konfigurieren der Socket-Bibliothek. Es kann sich z.B. um ein nachträgliches Ändern der Baudrate zwischen der Socket-DLL und dem CANopen Gerät handeln. Die Parameter, die an ConfigSocket übergeben werden, sind abhängig von der jeweiligen Library.
- Das Kommando ***Socket*** wird benutzt, um ein Socket-internes Kommando auszuführen. Diese internen Kommandos sind ebenfalls Socket-spezifisch und demnach abhängig von der jeweiligen Library.

5.4.3 Device Commands

Die DeviceCommands beinhalten prinzipiell alle Kommandos, die an das Modul *CO_RemoteCommand* geschickt werden können. Der Unterschied besteht darin, dass die Befehle in der CANopenConsole in Textform und nicht als Zahlencode eingegeben werden. Zur Unterscheidung zwischen den Geräten wird die Gerätenummer als Parameter nach jedem Befehl erwartet. Bei allen Kommandos entspricht die Gerätenummer Null einem Broadcast, d.h das Kommando wird an jedes initialisierte CANopen Gerät des Netzwerkes geschickt. Das kann von Vorteil sein, wenn Informationen über Objekte benötigt werden, die in allen Netzwerkgeräten definiert sind. So könnte z.B. das Kommando:

od 0 0x2001 0

folgende Ausgaben machen:

DEV1: OD[0x2001][0x00]: OT=7 DT=4 Dr- 0x00001234 = 4660
DEV2: OD[0x2001][0x00]: OT=7 DT=4 Dr- 0x00008765 = 34661
DEV3: OD[0x2001][0x00]: OT=7 DT=4 Dr- 0x00008765 = 34661

Der Präfix *DEV1:* wurde von der CANopenConsole hinzugefügt, um die Ausgabe dem richtigen Gerät zuordnen zu können. OD[0x2001][0x00] kennzeichnet den MUX des Objekts. OT=7 definiert den *Objecttype* – in diesem Fall *OT_VARIABLE*. DT=4 legt den Variablentyp auf *Integer32* fest. Die folgenden drei Zeichen Dr4- geben Auskunft über die Zugriffsrechte. Danach folgt der Inhalt des Objekts, dargestellt in hexadezimaler und dezimaler Schreibweise.
Genauere Informationen zur Deutung der Ausgaben des Kommandos od können über das Konsolenkommando ot angefordert werden. Für das gezeigte Beispiel ist die Ausgabe:

ot 7 4 Dr-

Object type	: 07	= OD_VARIABLE
Data type	: 04	= ODDT_INTEGER32
Storage type	: D	= Data stored in ROM and directly in object structure without ptr
Read attribute	: r	= Data can be read
Write attribute	: -	= Data can not be written

5.4.4 System Commands

Alle Kommandos, die in der CANopenConsole nicht bekannt sind, werden automatisch als Systemkommandos identifiziert und ausgeführt. Auf diese Weise ist der Entwickler in der Lage, seine Skripte dem Betriebssystem anzupassen, was zwar nicht dem Ziel der Plattformunabhängigkeit entspricht, sich aber manchmal nicht vermeiden lässt.
Sollte es Systemkommandos geben, die mit dem Namen eines internen Kommandos übereinstimmen, ist es möglich, die Ausführung über das Kommando system zu erzwingen.

5.5 Fernwartung

Die CANopenConsole gestattet dem Entwickler bereits den Zugriff auf alle CANopen-Geräte des Netzwerks. Zur Programmierung des Netzwerks ist das völlig ausreichend. In der Praxis reicht dieser Ansatz dennoch nicht aus. Dort wird die CANopen Applikation üblicherweise bei einem Kunden installiert. Der Kunde könnte - gerade im export-orientierten Deutschland - aus dem Ausland stammen. Tritt bei diesem Kunden ein Problem im Netzwerk auf, sei es durch Bedien- oder Softwarefehler, wird die Korrektur durch Anfahrtswege und Personalausfall sehr teuer. Eine Fernwartung des Netzwerkes wäre dagegen preiswert und ist relativ einfach zu realisieren.
Das TCP/IP Protokoll bietet hervorragende Mittel, um dies zu ermöglichen – den *Telnet Service* und die *SSH* (Secure Shell). Diese Dienste sind plattformunabhängig. Auf den kommunizierenden Rechnern können verschiedene Betriebssysteme laufen. Grundsätzlich ist die SSH vorzuziehen, da sie Daten verschlüsselt überträgt, während Telnet alle Eingaben im Klartext sendet – sogar Passwörter. Mit diesen Diensten ist es möglich, sich in entfernte Computer einzuloggen und dort Programme auszuführen, z.B. die CANopenConsole, die Zugriff auf alle Netzwerkgeräte bietet. Zu beachten ist allerdings, dass TCP/IP kein echtzeitfähiges Protokoll ist und daher die Fernwartung nur eingeschränkt im Konfigurationsmodus möglich ist.
Die besondere Eigenschaft dieser Dienste besteht darin, dass sie rein textbasiert sind – ein weiterer Grund, warum es sich bei der CANopenConsole um eine Konsolenanwendung handelt. Eine Ausweitung der CANopenConsole auf graphische Benutzeroberflächen wäre mit einer Festlegung auf eine GDI verbunden, z.B. MFC bzw. QT, und damit zusätzlichen Kosten sowie einer Einschränkung an Flexibilität.

6 Die Beispiel-Applikation

Um die Funktionsfähigkeit der CANopen-API zu testen, wurde eine einfache Beispielapplikation entwickelt, das ***CANopenExample***. Die benutzten Quellcodes zu diesem Projekt befinden sich im Verzeichnis sources/CANopenExample.
Diese Applikation besteht aus 2 Kameras der Firma OPTOLOGIC, auf denen ein C166 Mikrocontroller und ein SJA1000 CAN-Controller arbeitet. Als drittes Gerät fungiert ein Standard-PC mit CAN-Karte der Firma IXXAT. Die Kameras und der PC sind physikalisch durch eine Zweidrahtleitung, den CAN-Bus, verbunden. Zusätzlich sind die Mikrocontrollersysteme über eine serielle Verbindung mit dem PC verbunden, um Downloads der Programme durchführen zu können.

6.1 Das Master-Gerät

Das Master-Gerät ist in diesem Beispiel der PC, der die Kontrolle über das Netzwerk ausführt. Er ist der NMT-Master und hat die Aufgabe, das Netzwerk zu booten und in einen funktionsfähigen Status zu überführen. Es besitzt die Gerätenummer 1. Die Kommunikation mit dem CAN-Bus wird über die CAN-API abgewickelt, die als Hardware die CAN-Karte von IXXAT verwendet.
Die komplette Geräteanwendung ist in der Bibliothek *Camera1.dll* kodiert. Diese Library kann von den CommandSocket-Libraries des CommandManagers geladen werden, wenn die CANopenConsole als übergeordnete Anwendung benutzt wird.

6.2 Die Slave-Geräte

Alle Kameras im Netzwerk bilden die NMT_Slaves, d.h. sie sind dem Reglement des NMT_Masters unterworfen. Sie besitzen Gerätenummern, die aufsteigend bei 2 beginnen.
Beim Booten eines Gerätes geht es innerhalb der Funktion CO_InitCANopen() in den Status NMT_INITIALISATION über. In diesem Stadium werden grundlegende Initialisierungen wie die des Object Dictionaries vorgenommen. Nach erfolgreichem Abschluß wird automatisch in den Status NMT_PREOPRATIONAL übergegangen und auf Anweisungen vom NMT_Master gewartet. Diesem obliegt es, die Geräte in den Status NMT_OPERATIONAL zu setzen. Sobald der NMT_Master das Kommando sendet, beginnen die Slave-Geräte mit der Aufnahme von Messwerten (s. Kap. 6.4).

6.3 Booten des Netzwerks

Als übergeordnete Steuerungsapplikation wird die CANopenConsole verwendet. Die Konsole startet durch Ausführen des Scripts *CANopenExample/Scripts/CANopenExampleSession.cas*. Dieses Skript ordnet den Geräten die Kommunikationsbibliotheken zu. Der Master-PC wird durch die CommandSocketVCIPCI.dll gesteuert, die einen eigenen Thread startet und intern die Bibliothek Camera1.dll lädt. In dieser DLL ist die CANopen-Implementierung für den Master-PC enthalten.
Das Initialisieren des Netzwerkes beginnt mit dem Aufruf CO_InitCANopen() in der Library des Master-PC's. Während der Initialisierung sendet der Master einen NMT_Request an alle Teilnehmer des Netzwerkes, einen Reset ihrer Kommunikationschicht durchzuführen

(NMT_RESETCOMMUNICATION). Danach beginnen die grundlegenden Geräteinitialisierungen in der Geräteanwendung des Masters, z.B. Anlegen des Object Dictionaries. Im Anschluß erfolgt die automatische Überführung in den Status NMT_PREOPERATIONAL. In diesem Modus lädt der Master von jedem Teilnehmer den Kameraidentifier und die Beschreibung der Kameras herunter. Damit ist gleichzeitig sichergestellt, dass alle Teilnehmer diesen Status erfolgreich erreicht haben. Tritt nach dem dritten Versuch eines Downloads immer noch ein Fehler auf, ist das entsprechende Gerät nicht bereit. Die Kommunikation wird daraufhin abgebrochen - das Booten des Netzwerkes ist gescheitert. In diesem Fall wird menschliche Hilfe benötig.
Sind die Downloads dagegen erfolgreich, wird im Anschluss ein NMT_Request verschickt, um die Geräte in den Status NMT_OPERATIONAL zu setzen. Ist dieser Modus erreicht, ist die Initialierung beendet und die Funktion CO_InitCANopen() kehrt fehlerfrei zurück.

Bei allen externen Geräten wird nach erfolgreichem Abschluß der Funktion CO_InitCANopen() eine Endlosschleife betreten, die dem periodischen Pollen der Nachrichtenwarteschlange und dem Aufruf der Funktionen zur Nachrichtenauswertung dient.
Der NMT_Master kehrt nach erfolgreicher Ausführung der CANopen-Initialisierung zu den Wartefunktionen der CommandSocketVCIPCI Kommunikationsbibliothek zurück. Er betritt keine Endlosschleife. Dies würde das Multitasking empfindlich und unnötig stören. Wenn die PCI-Karte CAN-Nachrichten empfängt, wird ein Callback in der CAN-API aufgerufen, der einen Event an den Thread der Socketbibliothek schickt. Aufgrund des Events ruft der Thread eine Funktion zur Nachrichtenverarbeitung in der CANopen-Library Camera1.dll auf.

6.4 Test der CANopen-Anwendung

Für den Echtzeitbetrieb von Anwendungen ist es notwendig, Kenntnis von den Leistungsgrenzen des Systems und der Software zu haben. Dazu sind umfangreiche Tests nötig, die sich in zwei grosse Abschnitte untergliedern lassen, nötig.

Funktionsnachweis der Komponenten

In diesem Test geht es um die prinzipielle Funktionsfähigkeit einzelnen Komponenten und um das Zusammenspiel der Komponenten während einer Übertragung über das Netzwerk.
Der Nachweis der korrekten Funktionsweise einzelner Komponenten wird durch Simulation, z.B. durch Erzeugen von Testumgebungen und Debuggen des Sourcecodes, nachgewiesen. Im Einzelnen wurde getestet:

- Übertragung von SDO-Objekten unterschiedlicher Datenlängen
- Übertragung von PDO-Objekten mit unterschiedlichen Mapping-Einträgen
- Senden von SYNC-, Heartbeat-, Lifeguarding Nachrichten
- Auslösen von Callbacks
- Einbinden der CANopen-Anwendungen in verschiedene Systemumgebungen (z.B. Polling- bzw. Interrupt-Nachrichtenauswertung)
- Einbinden von unterschiedlichen Objekten (z.B. int, Time_of_Day, Remote Objects)
- Booten des Netzwerks

Dieser Test wurde für alle entwickelten Komponenten durchgeführt und erfolgreich abgeschlossen, d.h. alle Komponenten arbeiten entsprechend der Protokolldefinition bzw. den in Kap. 4 beschriebenen Schnittstellen.

Nachweis der Echtzeitfähigkeit unter Extrembedingungen

Nachdem die generelle Funktionsfähigkeit unter idealen Bedingungen nachgewiesen ist, muss sie auch unter Extrembedingungen getestet werden. Der Nachweis der Echtzeitfähigkeit gestaltet sich in sofern schwierig, dass die Echtzeitbetrachtung stark von der jeweiligen CANopen-Anwendung abhängig ist. In einigen Anwendungen treten die Echtzeitnachrichten in kurzen aber von allen Clienten gleichzeitig ausgelösten Nachrichten auf, in anderen treten sie als Burst eines Clienten auf. Aus diesem Grund ist eine Angabe über die Echtzeitfähigkeit der CANopen-Software nicht besonders aussagekräftig. Es müssen für jede Anwendung neue Testverfahren entwickelt werden. Um die häufigsten Anwendungsfälle abzudecken, wurden folgende Tests entworfen:

Abhängigkeit der Latenzzeit bei gleichzeitiger Übertragung von allen Geräten

Bei diesem Test soll ermittelt werden, wie die max. Zeitdauer einer Nachricht zu einem Zielgerät ist, wenn alle anderen Geräte gleichzeitig mit der Übertragung beginnen. Dazu ist es notwendig, die Auslösung der Übertragung zu synchronisieren, z.B. durch Tippen eines entprellten Schalters, der in allen Geräten einen Interrupt auslöst. In dem Zielgerät wird daraufhin eine Zeitmessung gestartet, die erst mit Eintreffen der letzten Nachricht endet. Dieser Test ist notwendig, da CAN-Nachrichten nicht sofort, sondern erst nach der Arbitrierung und erforderlicher Busübernahme verschickt werden können. Bei gleichzeitiger Übertragung ist davon auszugehen, dass die langsamste Nachricht, die mit der geringsten CAN-ID, mindestens die n-fache Zeit einer einfachen Nachrichtenübertragung benötigt. Die ermittelte Zeit kann als *worst-case* Zeit für diese Anzahl von Geräten angegeben werden, in der eine Nachricht garantiert beim Ziel eintrifft.

Abhängigkeit der Latenzzeit von der Burstgröße

Die im 1. Test ermittelte Zeit kann nur ein Richtwert sein, denn bei Auftreten von Burst-Nachrichten, ist diese Aussage nicht mehr korrekt. Angenommen, jede Anwendung sendet zwei PDO-Nachrichten direkt hintereinander an ein Ziel. Die Übertragung beginnt mit der Nachricht der höchsten CAN-ID. Im Anschluss wird die 2. Nachricht durch diese Geräteanwendung initiiert, wodurch Zeitverlust entsteht, der möglichweise zum Senden der zweithöchstpriorisierten Nachricht führt. Im Anschluss setzt Anwendung 1 seine zweite Nachricht ab. Aus diesem Planspiel folgt, dass sich im Burstfall die ermittelte Latenzzeit um die Anzahl der Burstnachrichten verdoppelt. Diese Abhängigkeit gilt es in einem Test nachzuweisen. Der Testablauf ist ähnlich dem des ersten Tests, allerdings wird bei diesem Test die Anzahl der zu sendenden Nachrichten pro Applikation und Testdurchlauf erhöht, um die Abhängigkeit eindeutig herauszustellen. Beim Anpassen von eigenen Applikationen kann später aus den gewonnenen Daten in Abhängigkeit von der zu erwarteten max. Burstlänge die max. Latenzzeit ermittelt werden.

Die Kamera-Anwendungen von OPTOLOGIC stellen typische Anwendungen für die Übertragung von Burstnachrichten dar. Bei der Überwachung von Förderbändern treten Fehler in der Regel in Bursts auf, sodass jede Kamera höchstwahrscheinlich mindestens einen Fehler an den Leit-PC senden wird.

Von den hier vorgestellten Tests ist lediglich der Funktionstest der Komponenten im Netzwerk komplett absolviert worden. Der Echtzeitnachweis konnte aufgrund fehlender Kameras noch nicht durchgeführt werden und muss in weiterführenden Arbeiten abgeschlossen werden. Es kann an dieser Stelle auch nicht der Anspruch auf Vollständigkeit der Testverfahren erhoben werden. In weiteren Untersuchungen müssen z.B. noch Nachweise zur Ausfallsicherheit und dem Betriebssystemeinfluss sowie des Laufzeitverhaltens der CANopen-API vorgenommen werden.

7 Auswertung

Aufbauend auf die CAN-API [1] wurde eine funktionsfähige CANopen Software entwickelt. Diese CANopen API entspricht dem CANopen Protokoll nach CiA Draft Standard 301 weitestgehend. Alle verbindlichen Dienste und eine Reihe optionaler Services werden unterstützt. Die in ANSI-C verfasste Programmierbibliothek ist portabel und bietet zusätzlich die Möglichkeit mehrere CANopen Instanzen gleichzeitig zu verwalten. Intern wird auf die dynamische Allozierung von Speicher verzichtet, wodurch einer Speicherfragmentierung vorgebeugt wird. Die API ist durch das Hinzufügen zusätzlicher Funktionsmodule in der Lage, Steuerkommandos zu empfangen und auszuführen.
Mit der CANopen API entstand eine Software, die einen guten Kompromiss zwischen den unterschiedlichen Programmierparadigmen von PC's und Mikrocontrollern darstellt. Trotz der sehr flexiblen Programmierung ist die API schnell und resourcensparend.
Die CANopen-API wurde auf Standard-PC's und eingebetteten Systemen, wie dem Kamerasystem von OPTOLOGIC, mit unterschiedlichen Compilern erfolgreich übersetzt und getestet. Der Test im echtzeit-kritischen Bereich konnte aufgrund fehlender Komponenten nicht komplett vollendet werden. Die Zusammenarbeit mit CANopen-Geräten anderer Hersteller kann zum heutigen Zeitpunkt nicht garantiert werden, da sowohl CANopen Konformitätstests sowie Geräte von Drittanbietern nicht vorhanden waren. Diese Tests müssen in weiterführenden Arbeiten durchgeführt werden.

Zusätzlich zu der CANopen Software wurde ein umfangreiches Administrierungsprogramm, die CANopenConsole, entwickelt. Mit Hilfe dieses Werkzeugs ist es möglich, komplette CANopen Netzwerke zu administrieren und zu konfigurieren. Zu diesem Zweck stellt das Programm eine Vielzahl von Befehlen zur Verfügung, die durch Skripte automatisiert werden können.
Die Verwendung von Kommunikationsbibliotheken entkoppelt die verwendeten Funktionsmodule voneinander und erlaubt die spätere Erweiterung auf andere Kommunikationswege, wie SDO-Tunnelung über CANopen oder USB.
Die Umsetzung als Konsolenanwendung ermöglicht zudem eine Ausführung in Telnet-Sitzungen, wodurch die Fernwartung des Netzwerkes ermöglicht wird.

Die Einsatzmöglichkeiten der CANopen-API in der Industrie sind äußerst vielfältig. Die Software ist keineswegs darauf optimiert, nur in Zusammenhang mit den Kamerasystemen der Firma OPTOLOGIC zusammenzuarbeiten. Der Einsatz der CANopenConsole empfiehlt sich nur für Geräte, die das Modul CO_RemoteControl enthalten und die dazugehörigen proprietären Befehle unterstützen.

Literaturverzeichnis

[1] Jan Blumenthal: Kleiner Beleg, *Entwicklung einer CAN-API*, 05/2001 Universität Rostock

[2] Ole Reinartz & Olaf Franke, Diplomarbeit, *Entwicklung einer CAN-Bus Applikation*, 1995, Universität Rostock

[3] Can in Automation e.V. (CiA): *CANopen Application Layer and Communication Profile, CiA Draft Standard 301 V4.0*, 1999, Erlangen

[4] Philips: *SJA 1000 Stand-alone CAN controller*, Nov. 1997

[5] Infineon: *16Bit Microcontrollers C165/C163 V2.0*, 1996

[6] Infineon: *16Bit Microcontrollers Instruction Set Manual*, 1995

[7] IXXAT: *Virtual CAN Interface V1.09*, 05/1998, Weingarten

[8] Uwe Koppe: *CANpie*, www.microcontrol.net

[9] Wolfgang Lawrenz: *CAN, Controller Area Network - Grundlagen und Praxis*, Hüttig Verlag 1997, 2. Auflage, ISBN: 3-7785-2263-9

[10] Konrad Etschberger: *CAN - Grundlagen, Protokolle, Bausteine und Anwendungen*, Carl Hanser Verlag München, Wien 1994,
ISBN: 3-446-17596-2

[11] Bosch: *CAN Specification V2.0*, 1991

[12] Keil Software: *Getting Started and Create Applications*, 03/1999, www.keil.de

[13] Keil Software: *Optimizing 166/167 C-Compiler and Library Reference*, 03/1998, www.keil.de

[14] Keil Software: *166/167 Assembler and Utilities*, 07/1996, www.keil.de

[15] Stock Microcomputersysteme: *SB01-CANopen Interface Board User's Manual*, 10/1997 Wolfratshausen

[16] STA Reutlingen: *CANopen Slave Example Source*, 01/1997

[17] Dr. Kreft: *Vorlesungsunterlagen Prozessautomatisierungsnetze*, 06/1999, Universität Rostock

[18] Microsoft: *MSDN Library*, 04/2000

A Compiler-Switches für die CANopen Software

Compiler-Schalter	Beschreibung
_DEBUG	Bei Aktivierung dieses Schalters werden eine Menge von Debug-Anweisungen während der Abarbeitung nach *stdout* mittels des TRACE() Kommandos ausgegeben.
_CO_NOCALLBACKS	CANopen führt keine ServiceCallbacks aus, die Aufschluß über Fortgang von Operationen geben, z.B. das ein SDO-Download erfolgreich war. Es erfolgt demnach keine Benachrichtung bei einer entfernten Anfrage nach Daten an die Applikation.
_CO_NOREADPDO	Der Service ReadPDO wird abgeschaltet. Dieser ist in der CANopen-Spezifikation als *optional* gekennzeichnet und ist nur für PDO-Master Geräte sinnvoll, die Daten von anderen Geräten anfordern. Das Gerät kann weiterhin PDO-Anfragen aus dem Netzwerk beantworten - es kann nur keine initiieren.
_CO_NOSDOCLIENTS	Der Client-Service Download/Upload SDO's wird abgeschaltet. Dieser ist in der CANopen-Spezifikation als *optional* eingestuft und ist nur für SDO-Master Geräte sinnvoll, die Daten von anderen Geräten anfordern. Das Reagieren auf SDO-Anfragen von fremden Geräten bleibt von diesem Schalter unbeeinflußt, d.h. das Gerät kann weiterhin SDO-Anforderungen beantworten.
_CO_NOMULTIPDOMAPPING	Bei dem Mapping von Objekten in PDO's handelt es sich um Bit-Mapping, d.h. Objekte werden bitweise gemappt, auch wenn die Länge Vielfachen von ganzen Bytes entspricht. Dieses Mapping erfordert viel Zeit, die in eingebetteten Systemen nicht vorhanden ist. Mit Hilfe dieses Schalters wird statt des Bit-Mappings eine Byte-Mapping verwendet, d.h. es werden immer nur ganze Bytes gemappt. Dies ist nicht ganz CANopen-konform, aber deutlich schneller.
_CO_REMOTECONTROL	Durch Einschalten dieses Switches wird das Softwaremodul zur Fernwartung mitkompiliert. Die Software wird dann über stdin/stdout fernwartbar.
_CO_INTCONTROL	Bei Setzen dieser Direktive werden Nachrichten vom Netzwerk bereits im Interrupt abgearbeitet. Ist dieser Schalter nicht gesetzt, erfolgt im Interrupt lediglich die Einordnung in Nachrichtenwarteschlangen, wobei die Abarbeitung im Hauptkontext erfolgt.
_CO_LIFEGUARDING	Die Aktivierung dieses Schalter fügt der Software das Lifeguarding-Protokoll hinzu, das vom NMT-Master benutzt wird, um den aktuellen Status des Netzwerkes zu überwachen. Auf die Benutzung des Schalters _CO_NOCALLBACKS sollte verzichtet werden.

_CO_HEARTBEAT	Die Aktivierung dieses Schalters fügt der Software das Heartbeat-Protokoll hinzu, das von jedem NMT-Slave benutzt wird, um den NMT-Master den aktuellen NMT-Status anzuzeigen. Auf Aktivierung des Schalters _CO_NOCALLBACKS sollte verzichtet werden, da sonst keine Auswertung der Heartbeatnachricht in der Applikation erfolgen kann.
_REDIRECT_PRINTF	Um in Ausgaben von Terminalanwendungen auf PC's eine Angabe über die Quelle mitauszugeben, müssen die Streams umgeleitet werden. Dies betrifft die Anweisungen TRACE() und _printf(). Die 'normalen' printf()-Kommandos werden nicht beeinflußt.

Tabelle 15 Compiler-Switches für die CANopen-Software

B Fernwartungskommandos

Remote Control	Kategorie	Erklärung
Set_Object_Dictionary	Beschreibung	Setzt einen Eintrag im Object Dictionary Parameter.
	Parameter	<ID> <SUBID> <data> [data]
	Beispiel	Setze Inhalt von Objekt [2][3]=5 1 2 3 5
Get_Object_Dictionary	Beschreibung	Gibt des Inhalt des spezifizierten Objekts aus.
	Parameter	<ID> <SUBID>
	Beispiel	Gibt Inhalt des Objekts[3][4] aus. 2 3 4
RPDO	Beschreibung	Die CANopen Software startet einen Read PDO-Request. Dabei muss sichergestellt sein, dass die PDO-Nummer im Gerät existiert und die Mapping sowie Communication Parameter im OD gesetzt sind.
	Parameter	<PDO-Nummer>
	Beispiel	Ein PDO-Request für PDO 1 wird gestartet 3 1
TPDO	Beschreibung	Die CANopen Software startet einen Transmit PDO-Request. Dabei muss sichergestellt sein, dass die PDO-Nummer im Gerät existiert und die Mapping- sowie die Communication-Parameter im OD gesetzt sind.
	Parameter	<PDO-Nummer>
	Beispiel	Ein Transmit PDO für PDO-Nummer 1 wird gestartet. 4 1
USDO	Beschreibung	Ein SDO Upload zu dem Zielgerät wird gestartet.
	Parameter	<ID> <SubID> <Zielgerät> <ZielID> <ZielSubID>
	Beispiel	Von Gerät 120 wird das Objekt[6][7] auf das Objekt [2][3] des eigenen OD geladen. 5 2 3 120 6 7
DSDO	Beschreibung	Ein SDO Download zu dem Zielgerät wird gestartet.
	Parameter	<ID> <SubID> <Zielgerät> <ZielID> <ZielSubID>
	Beispiel	Das Objekt [2][3] des eigenen OD's wird auf das Objekt [6][7] des Zielgerätes 120 übertragen. 6 2 3 120 6 7
RESET	Beschreibung	Das Gerät führt einen Reset aus.
	Parameter	-
	Beispiel	7

SHOW_OD	Beschreibung	Das komplette Object Dictionary wird nach *stdout* ausgegeben.
	Parameter	-
	Beispiel	8
BAUDRATE	Beschreibung	Die Baudrate des Gerätes wird neu gesetzt. Der Parameter spezifiert dabei die Nummer einer Übertragungsgeschwindigkeit (s. Can/Can.h).
	Parameter	<Modus>
	Beispiel	Es wird eine Baudrate von 1=20KB/s eingestellt. 9 1
NMT	Beschreibung	Setzt den NMT-Status des Gerätes neu. Falls kein Parameter angegeben wird, wird der aktuelle Status nach *stdout* ausgegeben
	Parameter	[Status]
	Beispiel	Das Gerät wechselt in den Modus "Pre-Operational" 10 127
SYNC	Beschreibung	Die Übertragung eines SYNC-Objects wird gestartet.
	Parameter	-
	Beispiel	11
HEARTBEAT	Beschreibung	Der Gerät überträgt einen Herzschlag (Heartbeat-Nachricht).
	Parameter	-
	Beispiel	12
LIFEGUARD	Beschreibung	Das Gerät startet eine Lifeguarding Übertragung zu dem angegebenen Zielgerät. Dieses sollte in kurzer Zeit antworten.
	Parameter	<Zielgerät>
	Beispiel	Ein Lifeguarding Prozess wird an Gerät 5 gesendet. 13 5

Tabelle 16 Kommandos zur Fernwartung

C Befehle der CANopen Console

Kommando	Beschreibung	Art
alias	Fügt der internen Kommandoliste ein Alias hinzu. *Syntax: alias <name>*	C
baudrate	Setzt die Baudrate des angegebenen CANopen Gerätes. Zu diesem Zweck gibt es vordefinierte Synonyme, die bei einem Aufruf von *Baudrate* ohne Parameter angezeigt werden. *Syntax: baudrate <Grätenummer> <Baudrate|Synonym>*	D
cd	Wechselt in das angegebene Verzeichnis. *Syntax: cd <Verzeichnis>*	Y
configsocket	Konfiguriert ein CommandSocket. Die Parameter sind abhängig vom Sockettyp. *Syntax: configsocket <Gerätenummer> <Parameter>*	S
dsdo	Startet ein DSO-Download auf dem angegebenen Gerät. *Syntax: dsdo <Gerätenummer> <ID> <SUBID> <ZielGerät> <ID> <SUBID>*	D
error	Gibt Informationen zu einem Fehlercode aus. *Syntax: error <Fehlernummer>*	C
exit	Beendet die CANopen Console. *Syntax: exit*	C
heartbeat	Auf dem angegebenen Gerät wird ein Heartbeat ausgelöst. *Syntax: heartbeat <Gerätenummer>*	D
help	Gibt die Hilfeseite der CANopen Console aus. Zusätzlich kann eine erweiterte Hilfe bei Angabe des Kommandos ausgegeben werden. *Syntax: Help [Kommando]*	C
info	Zeigt Informationen über die angemeldeten CANopen Geräte und die entsprechenden Sockets an. *Syntax: info*	C
initsocket	Initialisiert ein Socket und benutzt dazu die angegebene Library. Es können zusätzlich Parameter an den Socket übergeben werden. *Syntax: initsocket <Gerätenummer> <SocketBibliothek> [Parameter]*	S
lifeguard	Auf dem angegebenen Gerät wird eine Lifeguard-Anfrage gestartet. *Syntax: lifeguard <Gerätenummer>*	D
message	Analysiert eine CANopen Message im *pre-defined connection set* und gibt umfangreiche Informationen über den Inhalt der Nachricht aus. *Syntax: message <Identifier> [data]*	C
nmt	Setzt den NMT-Status des betreffenden Gerätes. Ist kein Status angegeben, wird der aktuelle Status des Geräts ermittelt. *Syntax: nmt <Gerätenummer> <NMT-Kommando>*	D
od	Setzt den Inhalt eines Eintrages des Object Dictionaries. Ist nur der MUX angegeben, wird der Inhalt des Objekts angezeigt. Wird lediglich die Gerätenummer angegeben, wird das komplette Object Dictionary ausgegeben. *Syntax: od <Gerätenummer> [ID] [SUBID] [Daten]*	D

ot	Gibt Informationen über den angegeben *Objecttype* zurück. Dies ist sinnvoll, um Ausgaben des Kommandos *od* richtig zu interpretieren. *Syntax: ot [objecttype] [datatype] [attributes]*	C
output	Ein- bzw. Ausschalten von Ausgaben des betreffenden Gerätes. *Syntax: output <Gerätenummer> <yes|on|1|0|no|off>*	C
quit	Beendet die CANopen Console. *Syntax: quit*	C
reset	Zurücksetzen eines CANopen Gerätes ohne Entladen der Socketbibliothek. *Syntax: reset <Gerätenummer>*	D
rpdo	Das betreffende Gerät führt ein PDO-Anfrage aus. *Syntax: rpdo <Gerätenummer> <PDO-Nummer>*	D
script	Das angegebene Skript wird ausgeführt. Wird kein Name angegebene, wird das zuletzt ausgeführte Skript nochmals durchlaufen. *Syntax: script [Skriptname]*	C
socket	Die Socket-Bibliothek führt das nachfolgende Kommando aus. Diese Kommandos sind socket-spezifisch und können hier nicht näher erläutert werden. *Syntax: socket <Gerätenummer> <Kommando> [Parameters]* *Beispiel:socket 1 download device1.h86*	S
system	Führt das nachfolgende Systemkommando aus. *Syntax: system <Kommandostring>*	Y
Sync	Auf dem angegebenen Gerät wird ein SYNC-Object erzeugt und übertragen. *Syntax: sync <Gerätenummer>*	D
tpdo	Das betreffende Gerät führt eine PDO Übertragung aus. *Syntax: tpdo <Gerätenummer> <PDO-Nummer>*	D
trace	Dient dem schrittweisen Ausführen von Skripten. *Syntax: trace <yes|on|1|0|no|off>*	C
unalias	Entfernt das angegebene *Alias* aus der internen Kommandoliste. *Syntax: unalias <Kommando>*	C
usdo	Startet ein SDO Upload auf dem angegebenen Gerät. *Syntax: usdo <Gerätenummer> <ID> <SUBID> <ZielGerät> <ID> <SUBID>*	D
wait	Die CANopen-Console wartet ein Vielfaches der angegebenen Zeit in 1/1000 sek. *Syntax: wait <Zeit>*	C
?	Gibt die Hilfeseite der CANopen Console aus. Zusätzlich kann eine erweiterte Hilfe bei Angabe des Kommandos ausgegeben werden. *Syntax: Help [Kommando]*	C
#	Einleitung eines Kommentars, der bis zum Zeilenende geht. *Syntax: # Kommentar*	C

Tabelle 17 Kommandos der CANopen Console

Art	Beschreibung
C	Console commands
D	Device commands
S	Socket commands
Y	System commands

D Beispiel eines Skripts für CANopen Console

```
####################################
#                                  #
# Script to test CANopen network   #
#                                  #
# project: CANopenExample          #
# author : Jan Blumenthal          #
# date   : 20.06.2001              #
#                                  #
####################################
cd e:\homes\eagleeye\Optologic\sources\CANopenExample\

@####################################
@#possible commands
@#script "DeviceCommands.cas"
@#trace on
@#output 1 on

@####################################
@# define important aliases
@####################################
@alias da="socket 0 download"
@alias d2="socket 2 download"
@alias d3="socket 3 download"
@alias d4="socket 4 download"

####################################
# Initialize cameras!
####################################
cd Programs
initsocket 2 CommandRS232.dll port=COM1 hexfile=Camera2.h86
initsocket 3 CommandRS232.dll port=COM2 hexfile=Camera3.h86
initsocket 1 CommandVCIPCI.dll

@####################################
@#configsocket 1 baudrate=9600
@#configsocket 2 baudrate=9600

@echo ------------------------------------------------------------------
@echo ---------- OPTOLOGIC CANopenConsole ----------
@echo ------------------------------------------------------------------

@info

####################################
# start testing
####################################
@#wait 2000
@echo if there was no error you will be able to start
@echo testing now

####################################
# set one entry of object dictionary
@od 1 0x2010 0 00 01 02 03 04 05 06 07 08 09 10
```

Erklärung

Ich erkläre, diese Arbeit selbstständig angefertigt und die benutzten Unterlagen vollständig angegeben zu haben.

Ort, Datum: Rostock, 31. August 2001