Ingenieurbüro für Echtzeitprogrammierung

CANopen® Master-Library

Dok-Rev. 1.1 vom

Software-Rev. 1.0 vom 29.02.2012

IEP GmbH • Am Pferdemarkt 9c • D-30853 Langenhagen • Tel.: +49 (511) 70832-0 • Fax: +49 (511) 70832-99 • E-Mail: info@iep.de
Web: http://www.iep.de

Inhaltsverzeichnis

1 Urheberrecht und Haftung....4
1.1 Handhabung....4
1.2 Erklärung....4

2 CANopen® Master-Library....5
2.1 Funktionalität....5

3 Funktionen der Library....6
3.1 Einmalig aufzurufende Funktionen....6
3.2 Zyklisch aufzurufende Funktionen....6
3.3 Optionale Funktionen....7

4 Beschreibung der Funktionen....8
4.1 canopen_master_version....8
4.2 canopen_syncpuls....8
4.3 canopen_config_master....8
4.4 canopen_start....9
4.5 canopen_SDO....9
4.6 canopen_stop....11
4.7 canopen_pre_cycle....11
4.8 canopen_pre_wait....12
4.9 canopen_master_cycle....12
4.10 canopen_post_cycle....12
4.11 canopen_node_start....12
4.12 canopen_node_stop....13
4.13 canopen_get_drv_error....13
4.14 canopen_get_emcy_message....15
4.15 canopen_node_err....15
4.16 canopen_io_state....15
4.17 Konfiguration....16
4.17.1 Datenstrukturen zur Konfiguration....16
4.17.2 PDO_map....16
4.17.3 m_datatyp....17
4.17.4 m_pdotyp....17
4.18 m_nodeinfo....18
4.19 Node Verbindungsüberwachung....19
4.19.1 Node Heartbeat....19
4.19.2 NodeGuarding....19

5 Beispielprogramm....20

5.1 Initialisierung der IO-Modul-Tabelle....20
5.2 Zyklische Anwendungstask (Single Master)....21
5.3 Multitasking Anwendung (MultiTask)....22

Revisionsliste:

Rev.	Datum	Na.	Änderung
1.0	24.02.2012	Kr	Erstellung
1.1	29.08.2012	Ko	Ergänzungen

1 Urheberrecht und Haftung

Alle Rechte an diesen Unterlagen liegen bei der IEP GmbH, Langenhagen.

Die Vervielfältigung, auch auszugsweise, ist nur mit unserer ausdrücklichen schriftlichen Genehmigung zulässig.

In Verbindung mit dem Kauf von Software erwirbt der Käufer einfaches, nicht übertragbares Nutzungsrecht. Dieses Recht zur Nutzung bezieht sich ausschließlich darauf, dass dieses Produkt auf oder in Zusammenhang mit jeweils **einem** Computer zu benutzen ist. Das Erstellen einer Kopie ist ausschließlich zu Archivierungszwecken unter Aufsicht des Käufers oder seines Beauftragten zulässig. Der Käufer haftet für Schäden, die sich aus der Verletzung seiner Sorgfaltspflicht ergeben, z.B. bei unautorisiertem Kopieren, unberechtigter Weitergabe der Software usw.. Der Käufer gibt mit dem Erwerb der Software seine Zustimmung zu den genannten Bedingungen. Bei unlizensiertem Kopieren muss vorbehaltlich einer endgültigen juristischen Klärung von Diebstahl ausgegangen werden. Dies gilt ebenso für Dokumentation und Software, die durch Modifikation aus Unterlagen und Programmen von IEP hervorgegangen ist, gleichgültig, ob die Änderungen als geringfügig oder erheblich anzusehen sind.

Eine Haftung seitens IEP für Schäden, die auf den Gebrauch von Software, Hardware oder Benutzung dieses Manuskriptes zurückzuführen sind, wird ausdrücklich ausgeschlossen, auch für den Fall fehlerhafter Software oder irrtümlicher Angaben.

Das Einverständnis des Käufers oder Nutzers für den Haftungsausschluss gilt mit dem Kauf und der Nutzung der Software und dieser Unterlagen als erteilt.

1.1 Handhabung

Lesen Sie bitte zuerst sorgfältig diese Dokumentation bevor Sie anfangen zu programmieren. Sie sparen Zeit und vermeiden Probleme.

1.2 Erklärung

Wir behalten uns das Recht vor, Änderungen, die einer Verbesserung der Schaltung oder des Produktes dienen, ohne besondere Hinweise vorzunehmen. Trotz sorgfältiger Kontrolle kann für die Richtigkeit der hier gegebenen Daten, Schaltpläne, Programme und Beschreibungen keine Haftung übernommen werden. Die Eignung des Produktes für einen bestimmten Einsatzzweck wird nicht zugesichert.

2 CANopen® Master-Library

2.1 Funktionalität

Die CANopen® Master-Library stellt Funktionen zur Verfügung, um mit CANopen® kompatiblen Geräten (Slaves) unter RTOS-UH kommunizieren zu können. Die Slaves werden vom Master konfiguriert und die Datenzuordnungen auf die eigenen Prozessvariablen wird durchgeführt.

Die CANopen® Master-Library übernimmt die komplette Verwaltung der angeschlossenen Slaves und regelt den Datenaustausch der Slaves mit dem Master (PDO-Verkehr).

Die CANopen® Master-Library ist Multitasking fest, d.h. die Libraryfunktionen können von beliebigen Tasks aufgerufen werden. Die eindeutige Zuordnung der im Master-Stack liegenden internen Verwaltungsstrukturen geschieht über ein Handle (hier ein void*), das bei allen weiter genutzten Funktionen übergeben werden muss.

Bevor die Funktionen der Library genutzt werden dürfen, muss die Konfiguration der angeschlossenen Slaves eindeutig festgelegt werden. Im Beispielprogramm `CANAPPL.C` zeigt die Funktion `configure_nodes`, wie die Slaves konfiguriert werden:

> Es wird festgelegt, wie viele Slaves sich am Bus befinden, unter welchen Knotenadressen sie zu finden sind und welche Werte mit ihnen ausgetauscht werden.

Implementierungsdetails des Master-Stacks:

- maximal 32 Devices in einer Master-Verwaltung
- 32 Rx- Buffers in der Verwaltung des Master-Stacks
- 32 Tx-Buffers in der Verwaltung des Master-Stacks
- 1 SDO-Kanal je Device für asynchrone SDO-Transfers

3 Funktionen der Library

Es gibt drei Arten von Funktionen in der Master-Library:

- einmalig aufzurufende Funktionen
- zyklisch aufzurufende Funktionen
- optional aufzurufende Funktionen

3.1 Einmalig aufzurufende Funktionen

Zu den einmalig aufzurufenden Funktionen gehören die Funktionen, die zur Konfiguration der Master-Library notwendig sind:

- `canopen_config_master`
- `canopen_start`

Diese Funktionen müssen mindestens einmal vor dem eigentlichen Start der CANopen-Zyklen aufgerufen werden.

3.2 Zyklisch aufzurufende Funktionen

Bei den zyklisch aufzurufenden Funktionen unterscheiden wir zwei Betriebsmöglichkeiten der Master-Library.

- ***Single-Task Anwendung***
  Abwicklung der CANopen-Zyklen in einer einzigen Task, in der der PDO Transfer für Lesen und Schreiben der Prozessdaten abgewickelt wird:
  - `canopen_pre_cycle`
  - `canopen_post_cycle`

- ***Multi-Task Anwendung***
  Hier wird das Lesen/Schreiben der Prozessdaten in getrennten Task durchgeführt, dieses hat den Vorteil, dass nicht aktiv auf das Eintreffen einer PDO vom Slave gepollt werden muß, sondern nur beim Eintreffen reagiert wird. Der CANopen® Zyklus der Task, die die Daten schreibt, muss aber weiterhin zyklisch laufen, damit die internen Timer der Master-Library upgedatet werden und die entsprechenden Timeouts überwacht werden können.

Lese-Task	`canopen_pre_wait`
Zyklus-Task	`canopen_master_cycle`
	`canopen_post_cycle`

3.3 Optionale Funktionen

Optional stehen noch weitere Funktionen zur Verfügung, die für den PDO-Austausch nicht zwingend notwendig sind:

- `canopen_master_version`
- `canopen_syncpuls`
- `canopen_SDO`
- `canopen_stop`
- `canopen_get_drv_error`
- `canopen_io_state`
- `canopen_node_err,`
- `canopen_get_emcy_message`

Achtung:

Alle einstellbaren Zeiten für den CANopen® Stack sind in der Einheit [ms] anzugeben. Die Genauigkeiten dieser Zeiten hängt von der Zykluszeit ab, mit der die zyklischen Funktionen aufgerufen werden.

4 Beschreibung der Funktionen

4.1 canopen_master_version

Diese Funktion gibt in einem Char-Pointer (char*) die aktuelle Versionsnummer des Master-Stacks zurück.

```
char* canopen_master_version( void ) ;
```

4.2 canopen_syncpuls

Die Funktion gibt ein `TRUE` zurück, wenn ein neues Sync-Telegramm vom Master erzeugt wurde. Damit kann die Konsistenz-Überwachung der empfangenen PDOs, die auf synchrone Betriebsart konfiguriert sind, stattfinden. Sie sollte im Zyklus direkt nach der Funktion `canopen_pre_cycle` bzw. `canopen_master_cycle` aufgerufen werden, da in diesen Funktionen der Status aktualisiert wird.

```
int canopen_syncpuls( void* handle ) ;
```

4.3 canopen_config_master

Diese Funktion dient zur Einstellung der Master-Funktionalität und der globalen Konfiguration des CAN-Busses.

Die Funktion konfiguriert den Master-Stack auf Grund der Informationen die in der Struktur `iom` abgelegt sind. Mit dem Aufruf werden alle zur Konfiguration der Slaves notwendigen SDOs und PDOs erzeugt und in der Master-Stack Verwaltung abgelegt. Es startet noch keine Kommunikation auf dem CAN-Bus, der Funktionsaufruf dient einzig zur Konfiguration des Master-Stacks.

Rückgabewert ist ein `void*` der den Handle auf die interne Verwaltungsstruktur enthält, dieser wird für alle nachfolgend beschriebenen Funktionen benötigt.

```
void* canopen_config_master( int baudrate ,
                             int channel ,
                             int used_nodes ,
                             int sync_time ,
                             int heart_beat_time ,
                             int MastID,
                             m_nodeinfo * iom
                           ) ;
```

Die Parameter bedeuten im Einzelnen:

baudrate	Baudrate des CAN-Bus [10, 20, 50, 125, 250, 500, 1000 KBit]
channel	zu nutzender CAN-Kanal [1..n]
used_nodes	Anzahl der angeschlossenen Slaves

MastID	Geräteadresse des Masters [1...127]
sync_time	SyncTime Intervall: wenn 0 → Es werden keine Sync-Telegramme gesendet. wenn != 0, werden vom Master Sync Telegramme in dem angegebenem Intervall gesendet.
heart_beat_time	HeartBeat Intervall: wenn 0 → keine Heartbeat Telegramme, dann ist auch keine Knotenüberwachung möglich. Wenn != 0, werden vom Master Heartbeat Telegramme in dem angegebenem Intervall gesendet und es ist eine Knotenüberwachung möglich.
iom	Die Konfiguration der Slaves

Beispiel: Bus mit 3 Knoten, 125 KBit auf CAN-Kanal 1, ohne Sync, 1 Sec Heartbeat, ID=1

```
void* mHandle = canopen_config_master( 125, 1, 3, 0, 1000, 1, &IO_MODULES[0]);
```

4.4 canopen_start

Die Funktion initialisiert das Hardware Interface für den CANopen®-Stack, sie gibt einen Returnwert zurück, der den Erfolg der Initialisierung angibt.

```
int canopen_start( void* handle ) ;
```

Return = 0, Hardware-Interface konnte nicht initialisiert werden, der Grund kann mit der Funktion *canopen_get_drv_error(handle)* abgefragt werden.

Return = 1, Initialisierung wurde erfolgreich durchgeführt.

4.5 canopen_SDO

Diese Funktion dient zum Abfragen/Setzen zusätzlicher Objekte im Slave, die außerhalb von PDO-Transfers realisiert werden sollen. Die Funktion wartet auf die Antwort des Slaves und gibt einen Returnwert zurück. Alle negativen Returnwerte bedeuten eine fehlerhafte Übertragung. Ein Returnwert von 1 zeigt eine erfolgreiche Übertragung an.

```
int canopen_SDO( void* handle,
                 int node,
                 int timeout,
                 int len,
                 void* val,
                 WORD Index,
                 BYTE SubIndex,
                 BYTE Modus ) ;
```

Beschreibung der einzelnen Parameter:

node - Nummer des Device [1...n]

timeout - Maximale Wartezeit in ms auf die Antwort

len - Anzahl der Datenbytes [1..4]

val - Pointer auf die Anwendungsdaten

Index - Objektindex im Device

SubIndex - Subindex des Objektes im Device

Modus - Type der Übertragung

Modus	Bedeutung
0x2F	1 Byte Daten schreiben
0x2B	2 Byte Daten schreiben
0x27	3 Byte Daten schreiben
0x23	4 Byte Daten schreiben
0x40	Lesen von Daten maximal *len* (<= 4) Byte
0x21	len Byte Daten Schreiben
0x41	len Byte Daten lesen

Returnwert	Fehler
>0	Kein Fehler
-1	Falsche *node* Nummer
-2	*len* < 0 angegeben
-3	Ungültiger *modus* angegeben
-4	*Modus* und *len* passen nicht
-5	Transfer wurde abgebrochen
< -5	DS-301 Fehlercode mit 0x8000000 gesetzt

DS-301 Fehlercode	Bedeutung
0x85030000	Toggle Bit not altered
0x85040000	SDO protocol timed out
0x85040001	Client/Server command specifier not known
0x85040002	Invalid block size
0x85040003	Invalid sequence number
0x85030004	CRC error
0x85030005	Out of Memory
0x86010000	Unsupported access to object
0x86010001	Read to write only object

0x86010002	Write to read only object
0x86020000	Object does not exists
0x86040041	Object can not map to PDO
0x86040042	Length/Number of objects exceed PDO
0x86040043	General Parameter reason
0x86040047	General internal incompatbility in device
0x86060000	Object access failed due to hw-error
0x86060010	Data type does not match len of parameter
0x86060012	Data type does not match, len too high
0x86060013	Daten type does not match. Len too low
0x86090011	Subindex does not exists
0x86090030	Value range of parameter exceeded
0x86090031	Value of parameter too high
0x86090032	Value of parameter too low
0x86090036	Maximum is less the minimum value
0x88000000	General error
0x88000020	Data can not be stored or transfered
0x88000021	Data can not be store or transfered local
0x88000022	Datan can not be store/transfer while deivce state
0x88000023	No object dictionary present

4.6 canopen_stop

Die Funktion ist nur beim Multitasking-Betrieb mit getrennter Lese-/Schreibtask notwendig, und muss beim Beenden des Masters aufgerufen werden, damit die LeseTask aus ihrem Warten auf ein Paket heraus kommt und sich auch beenden kann.

```
int canopen_stop( void* handle ) ;
```

4.7 canopen_pre_cycle

Diese Funktion muss zyklisch vom Anwenderprogramm aufgerufen werden, sie prüft, ob von den Slaves CANopen-Nachrichten empfangen wurden und bearbeitet sie entsprechend im Master-Stack. Mit dem Empfang von Input-Daten der Slaves werden die Input-Daten für das Anwendungsprogramm aktualisiert. Der Returnwert der Funktion gibt an, ob Input-Daten empfangen wurden.

```
int canopen_pre_cycle( void* handle ) ;
```

Wichtig:

Da alle internen Timer des Master-Stacks (Sync, Heartbeat, Inhibit-Event, NodeGuard) mit dem Aufruf dieser Funktion aktualisiert werden, sollte sinnvollerweise die Funktion mit einer kürzeren Zykluszeit aufgerufen werden, als der kürzeste Timer.

Die Genauigkeit der internen Timer hängt von dem zyklischen Aufruf dieser Funktion ab.

4.8 canopen_pre_wait

Im Multitasking-Betrieb muß diese Funktion von der Lese-Task aufgerufen werden, sie wartet bis eine Nachricht vom CAN-Bus eingetroffen ist. Beim Eintreffen entsprechender Input-Daten werden sie für das Anwendungsprogramm aktualisiert. Der Returnwert der Funktion gibt an, ob Input-Daten empfangen wurden.

Optional kann bei dieser Funktion eine Semaphore genutzt werden, um den Input-IO-Update mit dem Zyklus der Master-Task zu synchronisieren.

m_sema - Pointer auf Semaphore

m_sema	Funktionalität
= NULL	Keine Synchronisierung
!= NULL	Synchronisierung auf die Sema durchführen während der Input-IO-Update durchgeführt wird.

```
int canopen_pre_wait( void* handle , Sema* m_sema ) ;
```

4.9 canopen_master_cycle

Bei getrenntem Betrieb mit Lese-/Schreibtask muss diese Funktion zyklisch vom Anwenderprogramm aufgerufen werden, damit alle SDO-Transfer und Timer im Master-Stack richtig behandelt werden.

```
void canopen_master_cycle( void* handle ) ;
```

Wichtig:

Da alle internen Timer des Master-Stacks (Sync, Heartbeat, Inhibit-Event, NodeGuard) mit dem Aufruf dieser Funktion aktualisiert werden, sollte sinnvollerweise die Funktion mit einer kürzeren Zykluszeit aufgerufen werden, als der kürzeste Timer.

Die Genauigkeit der internen Timer hängt von dem zyklischen Aufruf dieser Funktion ab.

4.10 canopen_post_cycle

Die Funktion muss zyklisch vom Anwenderprogramm aufgerufen werden, sie überträgt alle vom Anwendungsprogramm geänderten Daten an die angeschlossen Slaves. Hierbei werden die Einstellungen aus der Konfiguration berücksichtigt, d.h. es werden die entsprechenden Timingparametern und Übertagungsarten verwendet.

```
void canopen_post_cycle( void* handle ) ;
```

4.11 canopen_node_start

Diese Funktion dient zum selektiven Starten eines Slaves [1...n], damit er an der Kommunikation teilnimmt.

```
int canopen_node_start( void* handle, int node ) ;
```

4.12 canopen_node_stop

Diese Funktion dient zum selektiven Anhalten eines Devices [1...n], dadurch werden alle PDOs des entsprechenden Slaves abgeschaltet.

```
int canopen_node_stop( void* handle, int node ) ;
```

4.13 canopen_get_drv_error

Diese Funktion liest den aktuellen Fehlercode aus dem Master-Stack aus, dieser wird jeden Zyklus aktualisiert. Es werden interne Hardware Fehler signalisiert.

```
unsigned int canopen_get_drv_error( void* handle) ;
```

Der Returnwert besteht aus einem Präfix, der die interne Funktion angibt und einer Fehlernummer zur Bezeichnung des Fehlers. Sobald das oberste Bit gesetzt ist, liegt ein Fehler vor:

Fehlercode = Präfix | Fehlernummer ;

Präfix	Ort des Auftreten (intern)
0x00000000	Kein Fehler
0x81000000	CAN - Feature Funktion
0x82000000	CAN – Write Funktion
0x84000000	CAN – Read Funktion
0x88000000	CAN – Init Funktion

Fehlernummer	Fehlerbeschreibung
0	Kein Fehler
1	CAN Baustein nicht verfügbar
2	Interner Warnlevel erreicht
3	Ungültiges Interface
4	Empfangspuffergröße ungültig
5	Kein CAN Init durchgeführt
6	CAN Off Bus Fehler
7	Overrun aufgetreten
8	Empfangspuffer übergelaufen
9	Reset, Read Puffer gelöscht
10	Identifier unzulässig
11	Länge unzulässig
12	Nicht unterstütztes Kommando
13	RTR unzulässig
14	Kein Speicher für Empfangspuffer
15	Transmit-Timeout aufgetreten
16	Frame-Format unzulässig
17	SAMP unzulässig
18	Baudrate unzulässig

4.14 canopen_get_emcy_message

Diese Funktion liest für einen Slave die letzte Emergency-Message aus. Dazu muss das Anwenderprogramm einen Puffer von 8 unsigned chars bereitstellen, indem die Message abgelegt wird.

slave	gibt den Index des in der Konfiguration angegebenen Slaves an
puffer	Puffer für die Meldung

```
void canopen_get_emcy_message( void* handle, int slave, BYTE* puffer );
```

Beispiel: Auslesen der EMCY-Message von Slave 3:

```
{
   BYTE e_message[ 8 ] ;
   ...
   canopen_get_emcy_message( handle, 3, e_message ) ;
   ..
}
```

4.15 canopen_node_err

Das Anwendungsprogramm muß ein Array für den Kurzstatus bereitstellen, in dem für jeden Slave ein Byte abgelegt werden kann. Beim Aufruf dieser Funktion wird der aktuelle Status jedes Slaves abgelegt. Wenn die Funktion ein FALSE liefert, arbeiten alle Slaves ohne Fehler. Meldet min. ein Slave einen Fehler, lautet der Returnwert 0x04. Der Bytepointer p muß auf das Array zeigen.

```
int canopen_node_err( void* handle, int used_nodes, BYTE* p ) ;
```

Bedeutung der Bits im Kurzstatus:

BitNr	Mask	Bedeutung
0	0x01	Device initialisiert
1	0x02	Device aktiv
2	0x04	Device meldet Fehler

Bei einem Returnwert von 0x04 kann mit Hilfe der Funktion `canopen_get_emcy_message` die letzte EMCY-Messages des Slaves abgefragt werden.

4.16 canopen_io_state

Diese Funktion aktualisiert das vom Anwenderprogramm bereitgestellte Array *io_st*. Dort werden die Stati der einzelnen Slaves abgelegt. Der Returnwert zeigt an, ob sich einer der Stati seit dem letzten Aufruf verändert hat, bei FALSE hat sich kein Status verändert.

```
int canopen_io_state( void* handle, int used_nodes, int* io_st ) ;
```

Achtung:

Der Status des Masters wird unter dem Index 0 in das Array *io_st* abgelegt. Die angeschlossen Slaves folgen in der Reihenfolge ihrer Konfiguration unter den Indices 1...n. Daher muss das Array *io_st* mindestens die Größe von *used_nodes* + 1 haben.

Die Bedeutung der Stati wird in folgender Tabelle angegeben.

Status	Bedeutung
-1	Wait for Bootup Message
0	undefiniert
1	Warte auf PreOperational
2	Reset SDO Kommunikation (PreOperational)
3	SDO Konfiguration läuft
4	Wait for Start Kommando
5	Operational , PDOs aktiv
97	Optionales Devices nicht vorhanden
98	Falscher DeviceTyp in der Konfiguration angegeben
99	Guard/Heatbeat Überwachungszeit abgelaufen

4.17 Konfiguration

Zur Konfiguration der CANopen® Master-Library muss als erstes die Datenzuordnung der einzelnen Slaves auf die eigenen Prozessvariablen und die Konfiguration der Datenübertragung festgelegt werden.

4.17.1 Datenstrukturen zur Konfiguration

Die Datenstrukturen sind in der `CANAPPL.H` beschrieben, mit Hilfe dieser Strukturen wird in der eigenen Applikation die Konfiguration vorgenommen.

4.17.2 PDO_map

Die Struktur `PDO_map` beschreibt einen Slave, sie gibt an unter welchem Index, Subindex und mit welcher Länge (in Bits) die Daten auf dem Slave zu finden sind. Die entsprechenden Werte sind in der Beschreibung oder EDS-Datei des Devices zu finden:

```
typedef struct PDO_map
{
   WORD   index    ;
   BYTE   subindex ;
   BYTE   len      ;
}
   PDO_map ;
```

4.17.3 m_datatyp

Die Struktur `m_datatyp` dient der Zuordnung zwischen den Slave-Daten und den Applikationsdaten. Unter `ioaddr` ist ein Pointer zuzuweisen, der auf die eigenen Applikationsdaten zeigt. `size` gibt die Länge des eigenen Datentypes (in Bits) an. Das Slave-Datum wird durch `obj_index` beschrieben.

Achtung:

Falls `size` und `obj_index.len` unterschiedlich sind, dann werden nur maximal `obj_index.len` Bits linksbündig unter `*ioaddr` abgelegt.

```
typedef struct m_datatyp
{
   int      size       ;
   BYTE *   ioaddr     ;
   PDO_map  obj_index  ;  /* Index im Objektverzeichnis  */
}
   m_datatyp ;
```

4.17.4 m_pdotyp

Die Struktur `m_pdotyp` beschreibt die Konfiguration der Datenübertragung (PDOs) eines oder mehrerer `m_datatyp` auf einer COB_ID der CANopen® Kommunikation.

```
typedef struct m_pdotyp
{
   int type          ;
   int cobID         ;
   int transtype     ;
   int inhibittime   ;
   int eventtime     ;
   int repeattime    ;
   m_datatyp io[64] ;
}
   m_pdotyp ;
```

Mit `type` wird die Richtung des Transfers festgelegt:

`TO_WRITE` gibt ein Schreiben auf das Device an

`TO_READ` gibt zu lesende Daten vom Device an

`cobID` ist die für die Kommunikation genutzte COB_ID, falls diese auf 0 gesetzt ist, wird vom Master-Stack automatisch eine COB_ID generiert:

Schreiben :	`0x200 + NODE_ID + 0x100 * pdo_index`	(für pdo_index = 0..3)
Schreiben :	`0x240 + NODE_ID + 0x100 * pdo_index`	(für pdo_index = 4..7)
Lesen :	`0x180 + NODE_ID + 0x100 * pdo_index`	(für pdo_index = 0..3)
Lesen :	`0x1C0 + NODE_ID + 0x100 * pdo_index`	(für pdo_index = 4..7)

Falls diese Defaulteinstellung nicht gewünscht ist, muß von der Applikation eine eigene `cobID` vorgegeben werden.

`io[ io_index ]` beschreibt ein einzelnes Slave-Datum, es können - da eine COB_ID maximal 64 Bits (8 Byte) lang ist - bis zu 64 Datenbeschreibungen angegeben werden.

z.B. Übertragung von 2 Variablen, die je 32 Bit lang sind, ergibt 64 Bit und die COB_ID ist voll.

`transtype` legt die Übertragungsart fest:

0	-	Synchrone Übertragung zu jedem Sync, wenn Daten geändert
1..240	-	Synchrone Übertragung bei jedem n-ten Sync
253	-	Übertragung auf Grund einer RTR-Anfrage
254	-	Asynchrone Übertragung bei Änderung des Werte

Optional können die Timer *inhibittime, eventtime, repeattime* bei der asynchronen Übertragung gesetzt werden, sie beeinflussen die Übertragung in folgender Weise:

inhibittime Die Daten werden auch bei schnelleren Änderung nur maximal in dieser angegebenen Intervallzeit übertragen.

eventtime Die Daten werden auch wenn keinerlei Änderung vorliegt, in diesem Intervall übertragen.

repeattime Die Daten werden in diesem Intervall übertragen

4.18 m_nodeinfo

Die Struktur `m_nodeinfo` dient zur Beschreibung eines Slaves für den Master:

```
typedef struct m_nodeinfo
{
   int bDoInit           ;
   int bOptional         ;
   int NodeID            ;
   int HeartbeatTime     ;
   int HeartbeatConsumer ;
   int GuardTime         ;
   int LifeTimer         ;
   m_pdotyp pdo[16] ;    /* Im Moment 16 PDOs / Modul */
}
   m_nodeinfo ;
```

`bDoInit` gibt an, dass der Master den Slave initialisieren soll.

Es sollte immer auf **1** stehen.

`bOptional` gibt an, ob der Slave notwendig für den Betrieb des Systemes ist.

`bOptional` = **0** : Der Bus nicht betrieben werden, wenn der Slave nicht reagiert.

`BOptional` = **1** : Es wird der Status 97 gemeldet, wenn der Slave nicht reagiert.

`NodeID` gibt die am Devices einzustellende Knotennummer an.

`pdo[x]` beschreibt die zu übertragenen Daten dieses Slaves und erzeugt jeweils ein PDO.

Die Anzahl der PDOs ist standardmäßig auf 8 RxPDO und 8 TxPDO eingestellt, damit ist die automatische Vergabe der cobID möglich. Falls mehr als 8 R/TPDOs genutzt werden sollen, so muss für die PDOs oberhalb des Achten zwingend eine eigene eindeutige cobID im `m_pdotyp` zugewiesen werden. Es können optional auch alle PDOs auf eigene cobID gesetzt werden.

4.19 Node Verbindungsüberwachung

Optionale Parameter zur Kommunikationsüberwachung sind die folgenden Elemente:

HeartbeatTime, HeartbeatConsumer, GuardTime, Lifetime

Zur Kommunikationsüberwachung unterscheidet der CANopen®-Master zwei verschiedene Methoden.

1. HeartBeat
2. NodeGuarding

Bei beiden Verfahren muss der Master für HeartBeat konfiguriert sein, sonst ist keine Kommunikationsüberwachung möglich.

4.19.1 Node Heartbeat

Wenn *HeartbeatTime* ungleich 0 gesetzt ist, erzeugt der Slave in diesem Intervall einen Heartbeat auf dem CAN-Bus. Dieses wird vom Master-Stack in der Art überwacht, dass in der 1.5 fachen Zeit mindestens ein Heartbeat des Devices empfangen werden muss. Ist dies der Fall, wird der Slave als OPERATIONAL gekennzeichnet, ansonsten wird der Slave mit dem Status = 99 signalisiert.

Wenn *HeartbeatConsumer* auf **1** steht, wird der Slave so konfiguriert, dass er den Heartbeat des Masters überwacht, bei Ausfall des Masters hängt die Reaktion vom Slave ab.

4.19.2 NodeGuarding

GuardTime gibt für den Slave das Prüfintervall für das Eintreffen eines Master-Heartbeats an.

Mit dem *LifeTimer* wird die obere Schranke für eine Fehlerreaktion des Slaves festgelegt, wenn bis zu diesem Zeitpunkt kein Master-Heartbeat eingetroffen ist.

5 Beispielprogramm

5.1 Initialisierung der IO-Modul-Tabelle

Zur Initialisierung der IO-Module-Tabelle ist es notwendig, die entsprechende EDS-Datei oder eine ausreichende Dokumentation für das anzuschließende IO-Module zu haben. Es sind die Bereiche 0x2000 ff. und 0x6000 ff. interessant, da hier die auszutauschenden Parameter beschrieben werden. In unseren Beispiel nutzen wir ein DS401 kompatibles digitales Ein-/Ausgabemodul.

In dem Beispiel wird von folgender Konfiguration ausgegangen:

1 Master

1 Slave (digital 8 Bit/IO Input/Output)

Der Bus soll mit Heartbeats überwacht werden, die Zykluszeit betrage 1 Sekunde.

Das digitale Modul ist nicht zwingend für den Betrieb notwendig, also optional.

Laut den EDS-Datei liegen die digital Ausgänge auf dem Objekt 0x6200 und die digitalen Eingänge auf dem Objekt 0x6000, sie sind jeweils 8 Bit breit. Die Übertragung der Daten soll asynchron mit einer Inhibit-Time von 100 ms erfolgen. Es sollen nur die Ausgänge beschrieben werden.

```
#include <CANAPPL.H>
#define MAX_NODES 1  /* 1 angeschlossenes Device */
m_nodeinfo IO_MODULE[ MAX_NODES ];
BYTE buffer[1] ; /* some application data buffer */

configure_nodes( void )
{
   m_nodeinfo* iom = &IO_MODULE[0] ;

   memclr( iom, sizeof( IO_MODULE ) ) ; /* alles auf 0 setzen */

   { /* Beschreibung für das Device  */
      iom->bDoInit       =    1 ; /* das Device soll initialisiert werden */
      iom->bOptional     =    1 ; /* das Device darf am Bus fehlen        */
      iom->HeartbeatTime = 1000 ; /* Sende all 1000 ms einen Heartbeat    */
      /* Datenübertragung für die Ausgänge beschreiben */
      iom->pdo[0].type        = TO_WRITE ; /* Daten zum Modul senden  */
      iom->pdo[0].transtype   = 254      ; /* asynchrone Übertragung  */
      iom->pdo[0].inhibittime = 100      ; /* alle 100 ms die Werteänderung senden */
      /* Datenverbindung für die Ausgänge beschreiben */
      iom->pdo[0].obj_index.index    = 0x6200 ; /* die Ausgänge lt. EDS -Datei */
      iom->pdo[0].obj_index.subindex =   1    ; /* siehe EDS-Datei             */
      iom->pdo[0].obj_index.len      =   8    ; /* 8 Bit datenbreite           */
      /* Datenverbindung für die Ausgänge der Applikation beschreiben */
      iom->pdo[0].io[0].size   =      8            ; /* hier 8 Bit Datenbreite    */
      iom->pdo[0].io[0].ioaddr = (BYTE*)&buffer[0] ; /* ptr aus Applikationsdaten */
    }
}
```

5.2 Zyklische Anwendungstask (Single Master)

Die Anwendungstask initialisiert den Stack und läuft dann in einer Endlosschleife mit einer Zykluszeit von 2 ms und verändert die Anwendungsdaten bei jedem Durchlauf:

```
   void

main( void  )
{
   int   i_cnt = 0  ;
   void *handle     ;
   int   used       ;
   int   chg_inputs ;

   used = configure_nodes( ) ;
   if ( handle = canopen_config_master( 125, 1, used, 0, 1000, 1, &IO_MODULE[0] ) )
   {
      if ( canopen_start( handle ) == 0 )
         printf(">>FATAL: CANAPP Init error %08X\n", canopen_get_drv_error( handle ) ) ;
      else
      {
         while( 1 )
         {
            chg_inputs = canopen_pre_cycle( handle  ) ;
            buffer[0] = (BYTE)i_cnt++  /* change application data */
            canopen_post_cycle( handle ) ;
            rt_resume_after( 2 ) ; // 2 msec Pause
         }
      }
   }
   else
      printf(">>FATAL: CANAPP No Memory to setup the Master \n") ;
}
```

5.3 Multitasking Anwendung (MultiTask)

Hier wird die CANopen® Anwendung auf zwei Task aufgeteilt, damit für die eingehenden Pakete nicht dauern gepollt werden muss.

Die 1. Task übernimmt die Verwaltungsaufgaben und die Ausgabe der Anwendungsdaten.

Die 2. Task liest die Pakete ein und ändert die zugehörigen Anwendungsdaten.

Die Anwendungstask initialisiert den Stack und läuft dann in einer Endloschleife mit einer Zykluszeit von 2 ms und verändert die Anwendungsdaten bei jedem Durchlauf.

Reader – Task :

```
#pragma TASK USE_FUNCTIONNAME

   Task*

M_READ( void* handle, int* changed, Sema* m_sema )
{
     int run = 1 ;
     unsigned long errcode ;

     while( run )
     {
         *changed  = canopen_pre_wait( handle, m_sema ) ;
         errcode = canopen_drv_error( handle ) ;
         if ( errcode == 0x84000009 )  /* canopen_stop wurde aufgerufen */
            run = 0 ;
     }
     rt_terminate_and_vanisch() ;
}
```

Master – Task :

```
   void

main( void  )
{
   int   i_cnt = 0  ;
   void *handle     ;
   int   used       ;
   int   chg_inputs ;
   int   synctime = 0 ; /* ohne Sync-Telegramme */
   Sema *m_sema   = 1 ; /* default ist free */

   used = configure_nodes( ) ;
   if ( handle = canopen_config_master( 125, 1, used, synctime, 1000, 1, &IO_MODULE[0] ))
   {
      if ( canopen_start( handle ) == 0 )
         printf(">>FATAL: CANAPP Init error %08X\n", canopen_get_drv_error( handle ) ) ;
      else
      {
         Task* r_task = M_READ( handle, &chg_inputs, &m_sema ) ; /* Reader starten */
         while( 1 )
         {
            rt_request_sema( &m_sema ) ;
            canopen_master_cycle( handle ) ;
            if ( synctime != 0 )
            {
               /* nur dann macht die Abfrage einen Sinn */
               if ( canopen_syncpuls( handle ) )
               {
                  /* Sync-Telegramm erzeugt, aktuelles Input Datenabbild konsistent */
               }
            }
            rt_release_sema( &m_sema ) ;

            buffer[0] = (BYTE)i_cnt++ ; /* change application data */
            canopen_post_cycle( handle ) ;
            rt_resume_after( 2 ) ; // 2 msec Pause
         }
      }
   }
   else
      printf(">>FATAL: CANAPP No Memory to setup the Master \n") ;
}
```