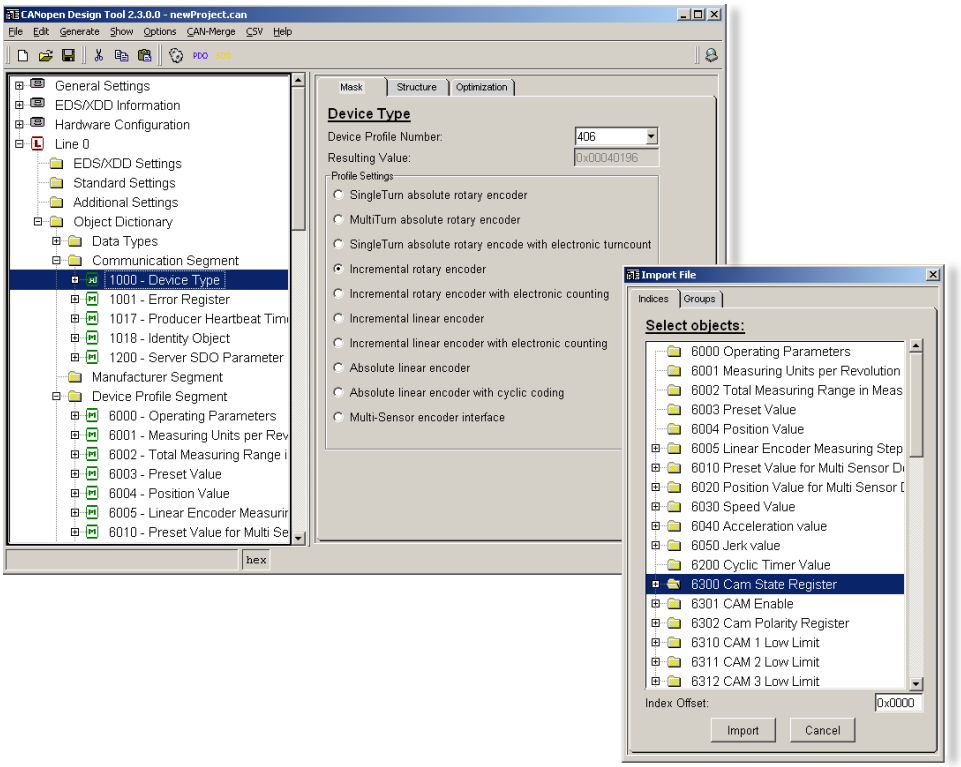
CANopen Library

User Manual

V4.5



Disclaimer
All rights reserved

The programs, boards and documentations supplied by *port* GmbH are created with due diligence, checked carefully and tested on several applications.

Nevertheless, *port* GmbH can not take over no guarantee and no assume del credere liability that the program, the hardware board and the documentation are error-free respective are suitable to serve the special purpose.

In particular performance characteristics and technical data given in this document may not be constituted to be guaranteed product features in any legal sense.

For consequential damages, which are emerged on the strength of use the program and the hardware boards therefore, every legal responsibility or liability is excluded.

port has the right to modify the products described or their documentation at any time without prior warning, as long as these changes are made for reasons of reliability or technical improvement.

We are thankful for hints of possible errors and may ask around for an information.

We will go all the way to verify such hints fastest

Copyright

© 2014 *port* GmbH
Regensburger Straße 7
D-06132 Halle
Tel. +49 345 - 777 55 0
Fax. +49 345 - 777 55 20
E-Mail service@port.de
Internet http://www.port.de

Table of Contents

1. Introduction

1.1. Applicability of the Documentation

The documentation of the CANopen Library by *port* consists of a user and a reference manual. The user manual serves as an introduction for using the CANopen Library. The procedure to use this CANopen Library and the process of integration into the customer's application are described here. This document and the reference manual describe the properties of all distributions of the CANopen Library. The CANopen Library source code is delivered in different configurations for multi or single CAN lines, master/slave or slave. Both user and reference manual are available as HTML documents.

The version/revision number of the user manual and the reference manual correlates to the version/revision of the CANopen Library software.

1.2. Product Overview

port is a member of the CAN in Automation (CiA). Our engineers are involved in the standardization activities in many of the Special Interest Groups of the CiA.

With this knowledge, *port* ensures that all CANopen products are conform to CiA standards. Because CiA does not certify software but rather CANopen devices, ask our support team for a reference of certified custom CANopen devices.

The CANopen Library by *port* has been developed with respect to the following points:

- CANopen functionality for both master and slave
- scalability to use only the kind and numbers of services the application needs
- independence of hardware and operating systems
- support of more than one CANopen network (up to 255 CANopen network lines)
- easy application interfaces
- very high portability and full ANSI-C conformity

All provided CANopen functionality fulfills the standards of CiA e.V., but not all optional functions are supported. Please contact our sales team for detailed information, see chapter 1.6.

The CANopen Library is an extremely flexible application. In control systems in which devices are used, implementing the CANopen technology can possibly infringe existing application patents. *port* GmbH should not held responsible for that.

1.3. System Requirements

The CANopen Library has been written in ANSI-C with a high degree of portability in mind. Therefore it is possible to compile the sources with any ANSI-C compliant compiler. The CANopen Library was tested with a lot of various compilers. The functions of the CANopen Library run on any system which guarantees:

- an interrupt handling for CAN, or available CAN device drivers
- hard- or software timer

For applications using an active CAN interface or providing CAN operating system drivers, a CAN interrupt handling is not necessary.

1.4. Additional Tools

The usage of the CANopen Library is support by the following tolls from *port*:

- CANopen Design Tool for the configuration and optimization of the CANopen Library, the configuration of desired hardware and the design of the object dictionary,
- CANopen Device Monitor for starting of CANopen communication in a CANopen network and
- CAN-REport for the analyzation of the CAN bus.

Demo versions of these tools are available on *port*'s web-page. The user manuals of these tools provides more details. The usage of the CANopen Design Tool is very recommended.

1.5. Installation

The CANopen Library inclusive the driver package are delivered:

for Windows™:
: with the installation program setup.exe or

for Linux:
: the source files.

All examples expect the recommended structure for successful compilation. Figure 1 shows the structure of the CANopen Library.

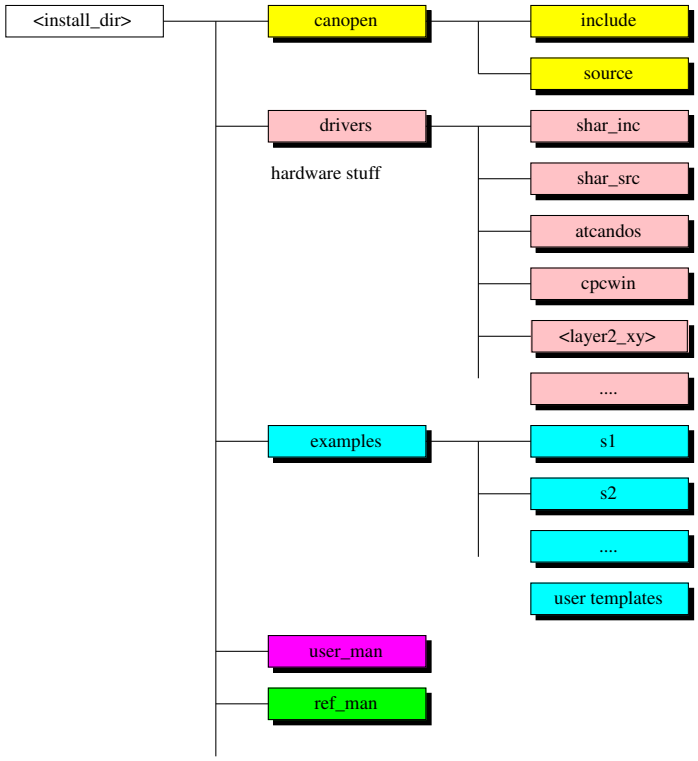


Figure 1, CANopen Library Directory Structure

Set the shift-width to 4 and tab-stop to 8 in your editor to get best performance while viewing the CANopen Library C files.

1.6. Support by *port*

port GmbH is one of the leading suppliers of CANopen communication technologies. The senior engineers at *port* support its customers by e-mail, by phone and by CAN/CANopen training courses. Additionally consultations in the entire field of CAN i.e. network planning, network configuration, message distribution, selection of devices and CANopen Device Profile implementations are available. Specific adaptations of our products according to customer requirements can be carried out on request.

Please contact us for sales via

e-mail: **service@port.de**

phone: **+49 345 777 55 - 0**

fax: **+49 345 777 55 - 20**

Please contact us for technical support via

e-mail: **support@port.de**

The engineers at *port* will give you some assistance as soon as possible.

2. CANopen Communication Model

CANopen is a set of existing and emerging profiles and was originally based on CAN Application Layer (CAL). CAL was the first available open application layer specification for CAN. Because CAL specifies a variety of data objects and services, the usage of these services was not easy. The CANopen Communication Profile comprises a concept to configure and communicate real-time-data as well as the mechanisms for synchronization between devices. Basically the CANopen Communication Profile describes how a subset of CAL services is used by devices. The restriction to a subset hereby reduces the amount of needed program memory to implement an open application layer.

Now all CANopen mechanisms and services are completely defined in the CANopen Application and Communication Profile.

The CANopen Device Profile describes the functionality of a particular device type and the communication with this device.

Two data types with different characteristics are dominating most automation networks and also CANopen. There are separate messages for process and service data. Furthermore CANopen defines an interface for data access. All data and parameter of a device, which should be visible from CAN, can be reached via the object dictionary.

2.1. Object Dictionary

All device parameters are stored in an object dictionary. This object dictionary contains the description, data type and structure of the parameters as well as the address from others point of view. The address is being composed of a 16 bit index and an 8 bit sub-index and guarantees therefore compatibility with existing device profiles (e.g. DRIVECOM). Only the CANopen specific entries have no correlation with other profile definitions. The sub-index refers to the elements of complex data types e.g. arrays and records (table 1).

Index	Sub-Index	Variable Accessed	Data Type
6092_h	0	Number of Entries	Unsigned8
6092_h	1	Baud Rate	Unsigned16
6092_h	2	Number of Data Bits	Unsigned8
6092_h	3	Number of Stop Bits	Unsigned8
6092_h	4	Parity	Unsigned8

Table 1, Organization of Complex Data Types

The following C-structure is the equivalent of the contents in table 1.

```
typedef struct {

    UNSIGNED8   NumberOfEntries;
    UNSIGNED16  BaudRate;
    UNSIGNED8   NumberOfDataBits;
```

```
    UNSIGNED8   NumberOfStopBits;
    UNSIGNED8   Parity;

} RS232_T;
```

The object dictionary is organized in different sections (table 2).

Index	Object
0000_h	not used
$0001_h - 009F_h$	Data Type Definitions
$00A0_h - 0FFF_h$	reserved
$1000_h - 1FFF_h$	Communication Profile Area (CiA-301)
$2000_h - 5FFF_h$	Manufacturer Specific Profile Area
$6000_h - 9FFF_h$	Standardized Device Profile Area (CiA-4xx), can be divided in in eight sections 800_h each, each containing objects of a different device profile
$A000_h - FFFF_h$	reserved

Table 2, Object Dictionary Structure

There is a range of mandatory entries in the dictionary which ensures that all CANopen devices of a particular type show the same behavior. The object dictionary concept serves for optional device features which means a manufacturer does not have to provide certain extended functionality on his device, but if he wishes to do so he has to do it in a predefined manner. Additionally, there is sufficient address space for truly manufacturer specific functionality. This approach ensures that the CANopen device profiles are future proof.

2.2. Service Data Objects

Service Data Messages, in CANopen called Service Data Objects SDO, are used for read and write access to all entries of the object dictionary of a device. Main usage of this type of messages is the device configuration. Besides reading and writing of the parameters and data, it is possible to download whole programs to the devices. SDOs are typically transmitted asynchronously. The requirements towards transmission speed are not as high as for PDOs. The SDO message contains information to address data in the device object dictionary and the data itself. Most existing profiles use 3 bytes to address objects, divided in two bytes for an index address and one byte for the sub-index address. Using the same scheme and considering one byte for the protocol four bytes remain for parameter data. Therefore a SDO transfer consists of a CAN message to initiate and perform data transfer and a CAN message for handshake.

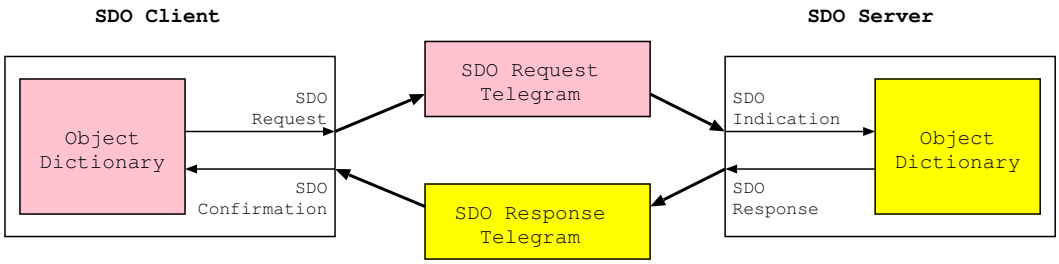


Figure 2, SDO Communication Principle

Figure 2 shows the communication between two devices. There are two variants for SDO usage. The first one is a write access and the second one a read access to the SDO server object dictionary. The SDO client initiates a write service with a SDO write request. The SDO server indicates the message, writes the value to the object dictionary and gives a response to the CAN network. The client gets a confirmation of that service. At a read request the confirmation message contains the data read. If it is necessary to transfer more than 4 byte, e.g. arrays or files, a sequence of segmented messages will follow the initiate transfer message, each one acknowledged by the data server. See figure 3 for the SDO protocol.

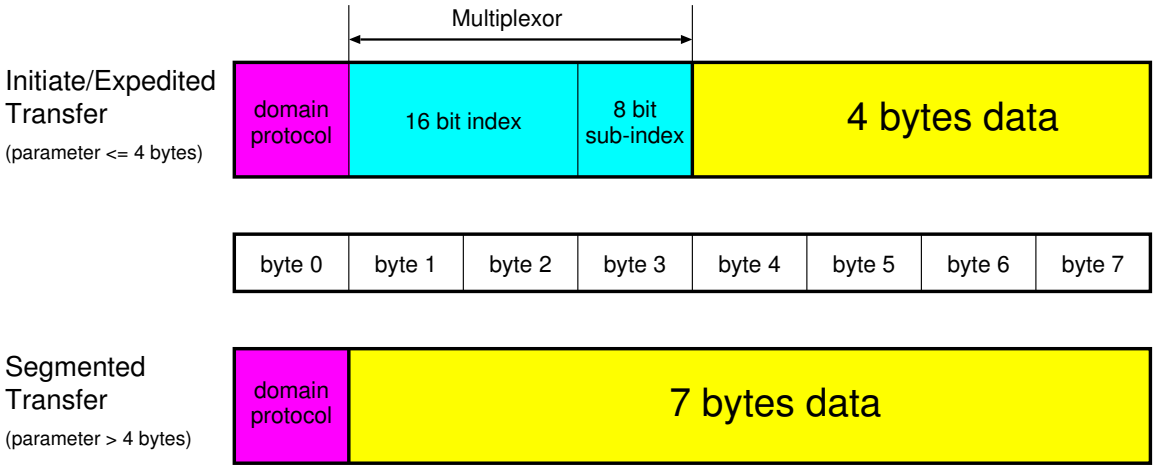


Figure 3, Multiplexed Domain Protocol for SDOs

Contrary to the PDOs the SDOs get low priority CAN identifiers. There are two device types for SDO handling. The first is the SDO server. These devices can not initiate a SDO service request. They can only react to a SDO indication. Such reactions are writing or reading values to/from the local object dictionary. The COB-IDs for the first server SDO are predefined and can not be changed in order to ensure the connection to the device. The COB-IDs are built like follow:

$$COB - ID_{TX} = 1408 + <node\text{-}ID> \quad (1)$$

$$COB - ID_{TX} = 580_h + <node\text{-}ID>$$

$$COB-ID_{RX} = 1536 + <node\text{-}ID> \tag{2}$$

$$COB-ID_{RX} = 600_h + <node\text{-}ID>$$

The equations are valid of the SDO server point of view. Every CANopen device must be an SDO server. The equivalent are the SDO clients. They initiate SDO services. Typical SDO client applications are configuration tools and control units. Each device can support up to 128 server SDOs and 128 s. The corresponding entries in the object dictionary are:

$$index_{server\ SDO} = 1200_h + (<server\ SDO\ number> - 1) \tag{3}$$

$$index = 1280_h + (< number> - 1) \tag{4}$$

The SDO parameter are organized in a structure (table 3). All entries can be changed besides the value for the first server SDO. The index 0022_h is only a reference to the structures type. It has to be replaced with the computed index mentioned above.

Index	Sub-Index	Field in SDO Parameter Record	Data Type
0022_h	0	number of supported entries	Unsigned8
0022_h	1	COB-ID client->server	Unsigned32
0022_h	2	COB-ID server->client	Unsigned32
0022_h	3	Node-ID of communication partner	Unsigned8

Table 3, SDO Parameter Structure

2.3. Process Data Objects

Process Data Messages in CANopen called Process Data Objects PDO are used to perform the real-time data transfer between different automation units. PDOs have to be transmitted quickly, without any protocol overhead and within a predefined structure.

For PDOs different transmission modes are distinguished (figure 4):

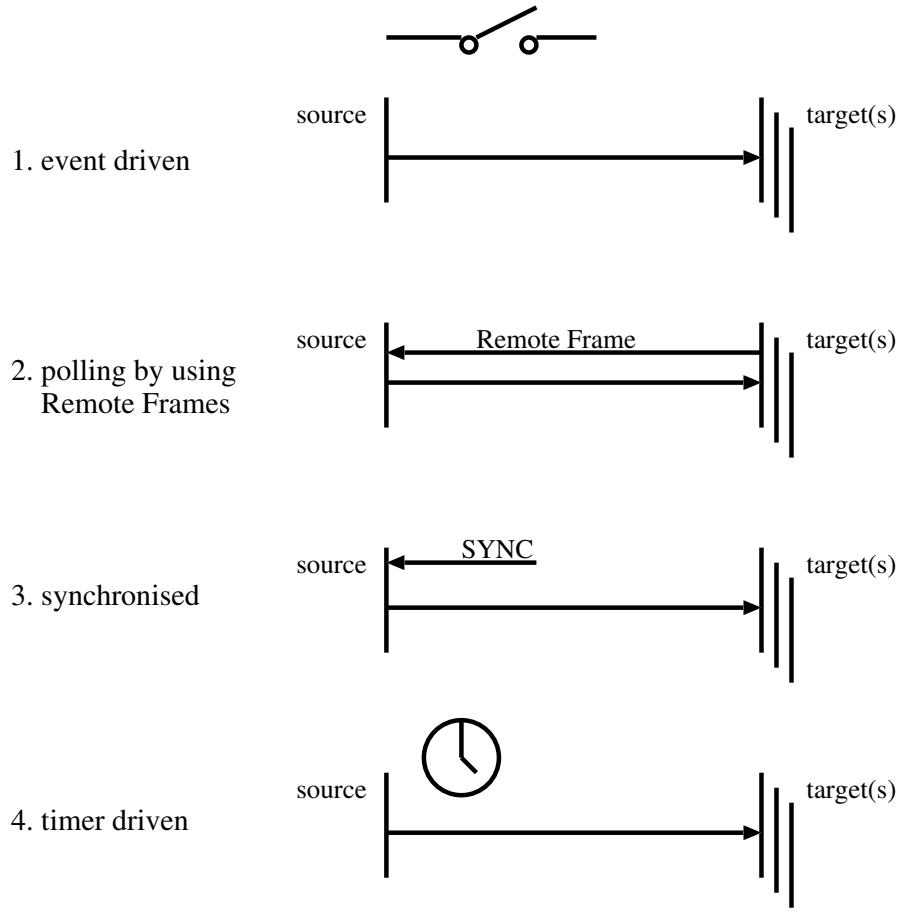


Figure 4, PDO Transmission Types

Asynchronous PDOs are sent on event (application/profile specific or timer) or on a remote request. Synchronous PDOs can be triggered cyclic or acyclic with the SYNC message.

An explicit confirmation for PDOs is not required. It is a CAN layer 2 message and carries no overhead. CANopen suggests a high priority in order to ensure their real-time behavior. CANopen defines PDO producer and PDO consumer. The producer sends PDOs and the consumer receives PDOs. Commonly a CANopen device can fulfill both properties.

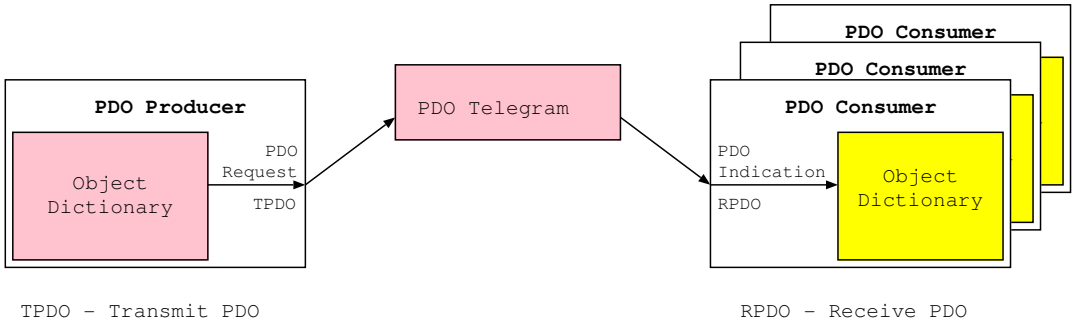


Figure 5, PDO Communication

The communication parameter of a PDO resides in the object dictionary. The indexes for PDOs are calculated as follows:

$$index_{RPDO-comm-par} = 1400_h + (<RPDO\ number> - 1) \tag{5}$$

$$index_{TPDO-comm-par} = 1800_h + (<TPDO\ number> - 1) \tag{6}$$

The range of the PDO numbers is 1..512. That means up to 512 Receive PDOs (RPDO) and up to 512 Transmit PDOs (TPDO) are possible for a device.
The communication parameters of PDOs are described with a structure (table 4). The index 0020_h is only a reference to the structures type. It has to be replaced with the computed index mentioned above. The sub-index 0 contains the number of implemented PDO parameters. Only sub-index 1 and 2 are mandatory. Subindex 1 describes the used COB-ID of the PDO. A detailed description is shown in table 6. A PDO communication channel between two devices is created by setting the TPDO COB-ID of the first device to the RPDO COB-ID of the second device. For PDOs a 1:1 and a 1:n communication is possible. This means that there is always only one transmitter, but an unlimited number of receivers. The transmission type (sub-index 2) describes the kind of transmission see table 5. For synchronous PDO it is possible to define a scaling factor. Transmission type 1 means PDO will be triggered with each SYNC object. If this entry has the value 240, the PDO will be sent/received with each 240th SYNC. The optional entry inhibit time defines a minimum time period between two PDO transmissions. This feature ensures that messages with lower priorities than the current PDO can be transmitted in the case of continuous transmission of the current PDO. The second optional parameter is only relevant for asynchronous Transmit PDOs. It defines an event timer. If the event time elapsed, the transmission of this PDO is started.

Index	Sub-Index	Field in PDO Parameter Record	Data Type
0020_h	0	number of supported entries	Unsigned8
0020_h	1	COB-ID	Unsigned32
0020_h	2	transmission type	Unsigned8
0020_h	3	inhibit time	Unsigned16
0020_h	4	reserved	Unsigned8
0020_h	5	event timer	Unsigned16

Table 4, PDO Communication Parameter Structure

Transmission Type	PDO Transmission
0	synchronous acyclic
1-240	synchronous cyclic
241-251	reserved
252	synchronous RTR only
253	asynchronous RTR only
254	asynchronous
255	asynchronous (standard device profile)

Table 5, Transmission Types

Bit Number	Value	Meaning
31(MSB)	0 1	PDO valid PDO not valid
30	0 1	RTR allowed RTR not allowed
29	0 1	11-bit ID 29-bit ID
28-11	0 X	if bit 29 = 0 if bit 29 = 1,COB-ID
10-0(LSB)	X	COB-ID

Table 6, COB-ID Code

The content of the PDO is encoded in the PDO mapping entries. A PDO can contain up to 64 single data elements from the object dictionary (in the case of 64 of the data are bits). The data are described via its index, sub-index and length. The mapping parameter of a PDO resides also in the object dictionary. The mapping indexes are built like follow:

$$index_{RPDO-map-par} = 1600_h + (<RPDO\ number> - 1) \quad (7)$$

$$index_{TPDO-map-par} = 1A00_h + (<TPDO\ number> - 1) \quad (8)$$

The index 0021_h is only a reference to the structures type. It has to be replaced with the computed index mentioned above. The sub-index 0 contains the number of mapped variables. The maximum of entries is either 64 or 8. This fact depends on the mapping granularity. (This is an feature of the CANopen Library implementation, not of the standard). Some devices support only byte-wise PDO mapping. The sub-index defines the order of the variables on the CAN telegram (PDO).

Index	Sub-Index	Field in PDO Mapping Record	Data Type
0021_h	0	number of mapped objects	Unsigned8
0021_h	1	1st object to be mapped	Unsigned32
0021_h	2	2nd object to be mapped	Unsigned32
0021_h	$64(40_h)$	64th object to be mapped	Unsigned32

Table 7, PDO Mapping Parameter

The entries from sub-index 1 contain a logical reference to the variables, which are to be transmitted/received (table 8). The date is described by its index and sub-index and its length. The length value represents the data's length in bits. Therefore it is possible to transmit only the relevant range of the data i.e. 3 bits of a C char value.

Index (16 bit)	Sub-Index (8 bit)	Object Length (8 bit)

Table 8, Structure of Mapping Entry

A special case of mapping is the so called dummy mapping. With this kind of mapping, it is possible to blind out irrelevant data. This feature is used for a 1:n communication, where each receiver utilizing only a part of the PDO. For dummy mapping the indexes 1-7 are used. These indexes are only references to data types. These entries are only space holders with the type corresponding size (table 9).

Index	Type	Length (bit)
0001_h	Boolean	1
0002_h	Signed8	8
0003_h	Signed16	16
0004_h	Signed32	32
0005_h	Unsigned8	8
0006_h	Unsigned16	16
0007_h	Unsigned32	32

Table 9, Indices of PDO Dummy Mapping Types

The mapping for the PDO can be static or changeable. If the mapping can be changed, it is called dynamic PDO mapping. Changing of mapping can be done in the state PRE-

OPERATIONAL (default) or OPERATIONAL. During OPERATIONAL state the configuration application is responsible for the data consistency.

2.4. Emergency Objects

The Emergency Message (EMCY) is a service, which signals internal fatal device errors. The error types are defined in the communication profile and the device profiles. The EMCY is transmitted with highest priority. CANopen defines EMCY producer and EMCY consumer. The producer transmits EMCYs and the consumers receive them. The EMCY telegram consists of 8 bytes. It contains an emergency error code, the contents of object 1001_h and 5 byte of manufacturer specific error code. Additionally an error handling exists. Each transmitted error code and the first two bytes of the manufacturer specific code will be pushed in the *predefined error field* on index 1003_h. This field can contain up to 255 error entries. The value of sub-index 0 shows the current number of entries. The most recently occurred error will be always inserted on the top of this field (sub-index 1). All older entries will be moved down. Are there no more errors on the device, an EMCY with error code 0 will be sent.

2.5. SYNC Objects

The SYNC object is a network wide system clock. It is the trigger for synchronous message transmission. The SYNC has a very high priority and contains no data in order to guarantee a minimum of jitter.

2.6. Time Stamp Objects

The Time Stamp object provides a common time reference. It is transmitted with high priority. The time is encoded in the type *Time of Day*. This data type contains the milliseconds since midnight and the number of days since January 1, 1984.

2.7. Error Control Mechanisms

For node monitoring two different mechanisms are defined. They are called **Node Guarding** and **Heartbeat**. Each device has to provide one of the error control mechanisms.

2.7.1. Node Guarding

The Node Guarding (Life Guarding) is the periodical monitoring of certain network nodes. Each node can be checked from the NMT master with a certain period (*guard time,* $100C_h$). A second parameter (*life time factor,* $100D_h$) defines a factor when the connection should be applied as lost.

The resolution of the guarding time is 1 ms. The condition for Node Guarding on a slave device is that *guard time* and *life time factor* are not zero. Guarding is started with the

first guarding telegram of the master.

During Node Guarding the master sends a RTR frame to each guarded slave. The slave answers with its current state and a toggle bit. This toggle bit alternates for each cycle.

➥ Node Guarding has a big influence on network load!

2.7.2. Heartbeat

Heartbeat is an error control service without need for remote frames. The Heartbeat producer transmits cyclic a Heartbeat message. One or more Heartbeat consumers receive this message and monitoring this indication. Each Heartbeat producer can use a certain period (*Heartbeat producer Time,* 1017_h). Heartbeat starts immediately if the Heartbeat producer Time is greater zero.

The Heartbeat consumer has to monitor the Heartbeat producer. For monitoring the Heartbeat consumer has an entry for each Heartbeat producer at its own object dictionary. The *Heartbeat consumer Time* (1016_h) can be different for each Heartbeat producer but should be greater than the Heartbeat producer Time. Usually the Heartbeat will be configured by the network configuration manager.

The resolution of the Heartbeat times is 1 ms.

➥ Heartbeat has a big influence on network load - but in effect, the half of the load of the Node Guarding monitoring!

2.8. Boot-up Message

After a CANopen node has finished its own initialization and entered in the node state PRE-OPERATIONAL it has to send the *Boot-up Protocol* -Message. This message indicated that the slave is ready for work (e.g. configuration). This protocol uses the same identifier as the error control protocol (Node Guarding or Heartbeat).

2.9. Network Behavior

Another simplification for CANopen is the definition of a minimal boot-up procedure for devices, shown in the state diagram in figure 6.

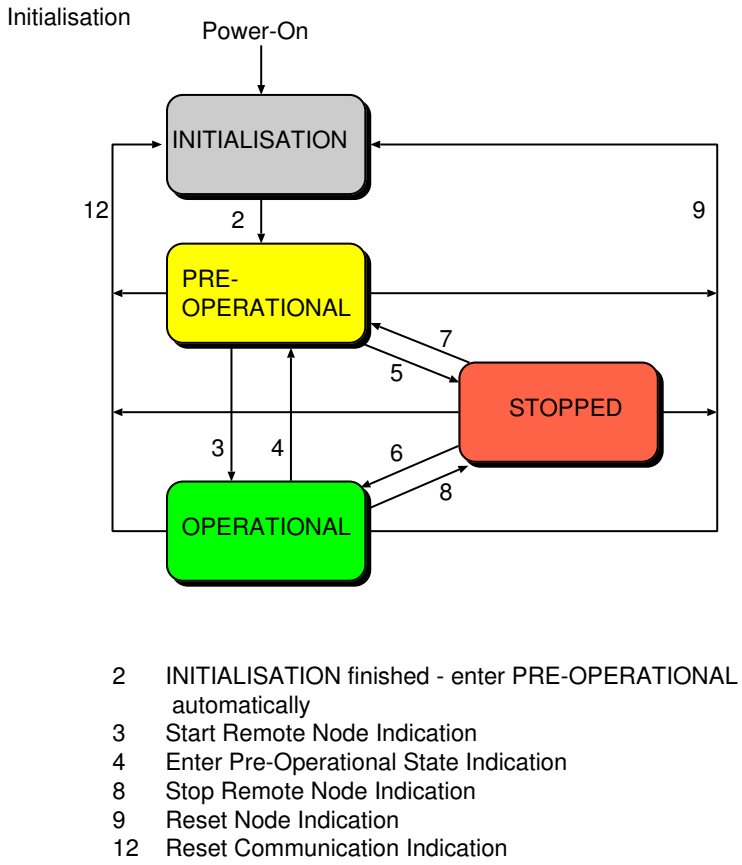


Figure 6, Minimal Boot-Up Procedure

Devices with the minimal boot-up procedure contain only three states PRE-OPERATIONAL, STOPPED and OPERATIONAL. The difference between master and slave devices is the initiation of the state transitions. The master controls the state transitions of each device in the network. After power-on a device is going to be initialized and set in the state PRE-OPERATIONAL automatically. In this state reading and writing to its object dictionary via the service data object (SDO) is possible. The device can now be configured. That means setting of objects or changing of default values in the object dictionary like preparing PDO transmission. Afterwards the device can be switched into the OPERATIONAL state via the command *Start Remote Node* in order to start PDO communication. PDO communication can be stopped by the network master by simply switching the remote node back to PRE-OPERATIONAL by using the *Enter Pre-Operational State* command. Via the *Stop Remote Node* command the master can force the slave(s) to the state STOPPED. In this state no services besides network and error control mechanisms are available. The command *Reset Communication* resets the communication on the node. All communication parameters will be set to their defaults. The

application will be reset by *Reset Node*. This command resets all application parameters and calls *Reset Communication* command. All needed NMT commands except the minimum boot-up use only CAN identifier 0. The commands are distinguished with a command specifier in the first data byte of the NMT message (table 10).
The second byte specifies the addressed node-ID. If the node-ID is zero the command is valid for all nodes in the network (broadcast).

NMT master Telegram: COB-ID 0		
Byte Number	Byte 0	Byte 1
Meaning	Command Specifier	Node-ID
Data Type	Unsigned8	Unsigned8

Table 10, NMT master Telegram

State/Service	SDO	PDO	EMCY	TIME	SYNC	NMT	ErrCtrl	Boot-Up
INIT	-	-	-	-	-	-	-	x
STOPPED	-	-	-	-	-	x	x	-
PRE-OPERATIONAL	x	-	x	x	x	x	x	-
OPERATIONAL	x	x	x	x	x	x	x	-

Table 11, Validity of CANopen Services

In order to reduce configuration effort for simple networks a mandatory default identifier allocation scheme is defined not only for NMT messages, also for the other services. These identifiers are available in the PRE-OPERATIONAL state directly after initialization (if no modifications have been stored) and may be modified by means of dynamic identifier distribution or SDO access (default way). A device has to provide the corresponding identifiers only for the supported communication objects.

The pre-defined master/slave connection set supports one emergency object, one server SDO, at maximum 4 Receive PDOs and 4 Transmit PDOs and the error control objects. The COB-ID is built from a function code, representing the object type, and the 7 bit module or node-ID. Table 12 shows a simplified version of what you can find in CiA-301.

Object	Function Code	COB-IDs	Index
broadcast objects			
NMT	0000_b	0	-
SYNC	0001_b	128	1005_h
TIME	0010_b	256	1012_h
peer-to-peer objects (node-ID related)			
EMCY	0001_b	129-255	1014_h, 1015_h
PDO1 (tx)	0011_b	385-511	1800_h
PDO1 (rx)	0100_b	513-639	1400_h
PDO2 (tx)	0101_b	641-767	1801_h
PDO2 (rx)	0110_b	769-895	1401_h
PDO3 (tx)	0111_b	897-1023	1802_h
PDO3 (rx)	1000_b	1025-1151	1402_h
PDO4 (tx)	1001_b	1153-1279	1803_h
PDO4 (rx)	1010_b	1281-1407	1403_h
SDO (tx)	1011_b	1409-1535	(1200_h)
SDO (rx)	1100_b	1537-1663	(1200_h)
NMT Error Control	1110_b	1793-1919	1016_h, 1017_h

Table 12, Function Codes for Default COB-IDs[1]

The resulting COB-ID for a object is built:

$$\textit{COB-ID} = ((\textit{function code}) * 128) + \textit{<node-ID>} \tag{9}$$

The default COB-ID allocation is only useful for peer to peer communication between a control application and the nodes. In order to use the advantages of CAN a COB-ID distribution after boot-up is necessary. The COB-IDs for the services SYNC, TIME, EMCY, PDO and SDO can be parameterized in the range of 1-1760. In this range only the COB-IDs of the used 1st SDO server is reserved. The order of priority should be SYNC, TIME, EMCY, synchronous PDOs, asynchronous PDOs and SDOs. The distribution can be done via SDOs.

2.10. CANopen Device Profiles

A device profile defines a standard device. For these standard devices a basic functionality has been specified, which has to exhibit every device within a class. The CANopen Device Profiles ensure a minimum of identical behavior for a kind of devices.

The layers of a device profile is shown in figure 7.

[1] The table has to be seen from the devices point of view.

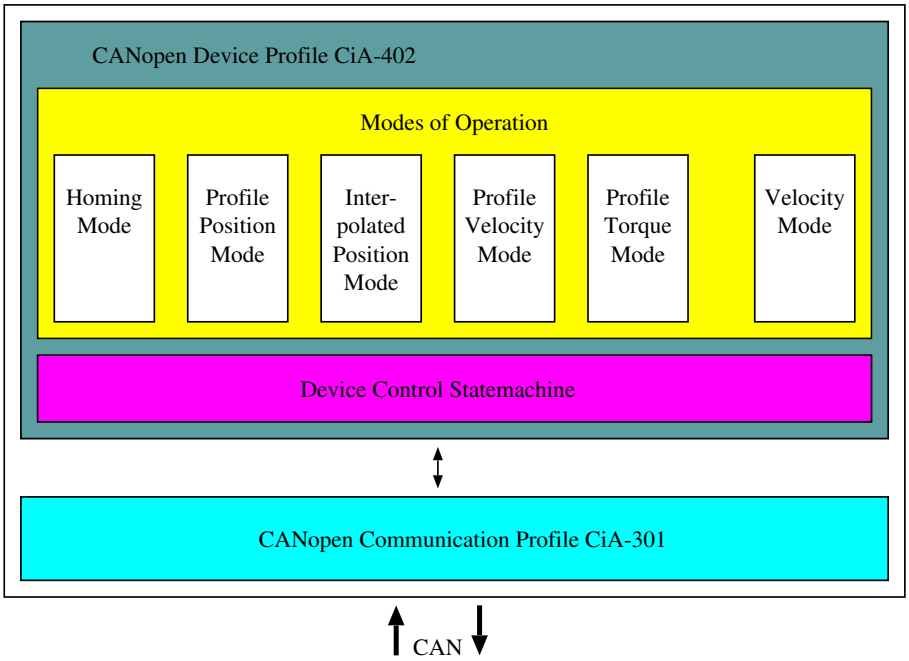


Figure 7, Communication Architecture for CiA-402

Each device has to fulfill the requirements on the behavior i.e. the implemented application state machine. Further it has to support all mandatory objects. These objects are parameter and data for the device. Additionally the manufacturer can decide about supported optional objects. All parameters and data, which are not covered by the standardized device profiles can be realized as manufacturer specific objects (2000_h - $5FFF_h$).

The constantly growing list of actual available profiles can always be found at *port*'s webpage.

Implementation of these device profiles can be done very easy by using the *CANopen Design Tool*. That tool provides databases with all objects for many of the standardized device profiles. Furthermore source code for realizing the CiA-401 and other functions are available.
Implementation of missing device profiles can be done very easy by using the *CANopen Design Tool* as data entry tool.

3. CANopen Library

3.1. CANopen Library Concept

The CANopen Library is offered at different configuration levels. All configuration levels are built upon one another. The functionality of the lower level is contained in the higher one.

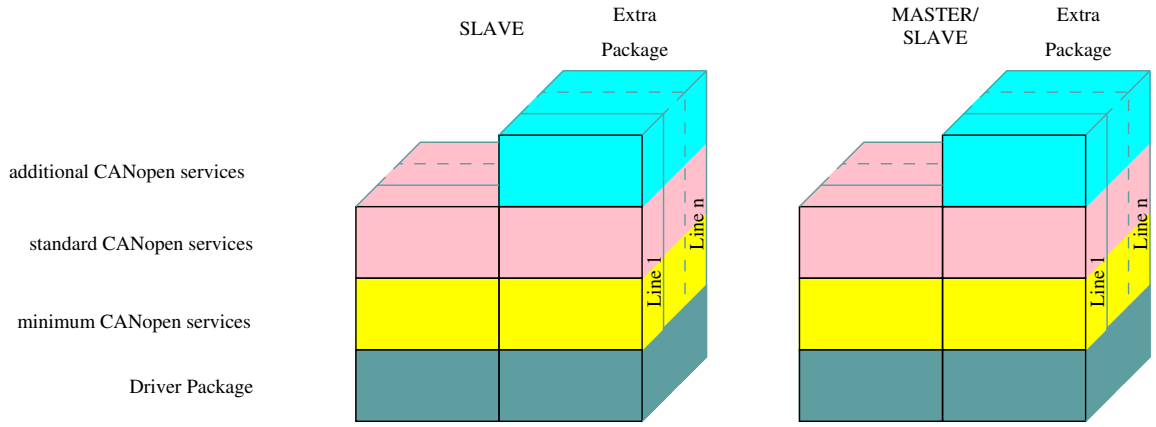


Figure 8, CANopen Library Grades

Not all services can be divided into master and slave services. Therefore some services are contained in the master and in the slave version.

The handling of services is the same for master and slave, so that a master can contain also slave functionalities. This means that a device, which is conceived as an I/O device can provide NMT master services in the network at the same time.

Table 13 shows the service attributes supported in the various configuration levels of the CANopen Library.

	Slave	Master/Slave	Extra
Service	Standard	Standard	Package
SDO server	128	128	
SDO client	128	128	
SDO Segmented Transfer	✓	✓	
SDO Block Transfer			✓
Dyn. SDO Slave			✓
SDO Manager			✓
Program Download	✓	✓	
PDO consumer	512	512	
PDO producer	512	512	
Dynamic Mapping	✓	✓	
Bit-wise Mapping	✓	✓	
MPDO Source Mode			✓
MPDO Dest. Mode			✓
Node Guarding master		✓	
Node Guarding slave	✓	✓	
Life Guarding	✓	✓	
Heartbeat consumer	128	128	
Heartbeat producer	✓	✓	
EMCY consumer	128	128	
EMCY producer	✓	✓	
TIME consumer	✓	✓	
TIME producer		✓	
SYNC consumer	✓	✓	
SYNC producer		✓	
NMT slave	✓	✓	
NMT master		✓	
NMT Flying Master			✓
NMT Boot-up Procedure		✓	
Configuration Manager		✓	
Safety Communication			✓
Redundancy Support			✓
LSS Services (CiA-305)	✓	✓	
LED Indicator (CiA-303-3)	✓	✓	
Nonvolatile Storage	✓	✓	
CiA-401 Framework			✓

Table 13, CANopen Library Service Attributes

All multi-line versions can serve one or more physical CAN lines. Each CAN line operates completely independently of the other lines. The network behavior (master/slave) can be different on every CAN line, i.e. each line can be initialized as network master or slave. Interaction between all the individual lines is possible.
Typical applications for multi-line versions are process data logger, human machine interfaces gateways, and devices which work with internal CANopen networks.

If it is necessary to implement additional properties (programmable device services, additional profiles, etc.), the user can enhance the CANopen Library with Extension Packages offered by *port* GmbH.

With all versions of the CANopen Library it is possible to use environments without an operating system, with single-tasking or multi-tasking systems with only rudimentary resource control mechanisms.

The CANopen Library consists of two main parts:

- CANopen protocol services
  CANopen protocol services according to the standards CiA-301, CiA-302
- Hardware driver
  access to the users target hardware (especially CAN controller).

The segmentation offers the following advantages:

1. Hardware-independence (CAN controller, micro controller or both) of the main part of the protocol stack.
   Exchange target hardware while retaining the CANopen functionality by only replacing the hardware driver layer for the new target hardware.
2. Comfortable development
   Development of the CANopen functionality can take place on easy to treat hardware (i.e. PC with CAN interface) and development environment, afterwards only the device driver for the hardware has to be exchanged.
3. Simple extensibility
   Extension of the devices functionality without necessary modifications of the driver interface.
4. Application with multi-tasking operating system
   The segmentation into several processes is already done by the separation into functional modules. Interprocess communication between driver and CANopen layer is already prepared.

The CANopen Library was developed in ANSI-C in order to obtain an easy portability. All hardware and operating system specific functions were separated in extra modules. These modules are the contents of the *CANopen Driver Packages*.

A further important point is the high scalability of the code size. The user can decide to compile a network master or slave device. Furthermore every kind of CANopen service is located in its own module e.g. **pdo.c**, **sdo.c**. Therefore the user is able to select only the required modules. Additionally it is possible to use compiler defines in order to select several CANopen Library properties. The advantage of that is that code size grows only with the used CANopen functionalities.
For an easy configuration of the CANopen Library the user is supported by the interactive

CANopen Design Tool.

The CANopen Library contains all CANopen services in respect to CiA-301 and the important services in respect to CiA-302 (table 13). For all of these many of the features have been implemented. From this wide range of possibilities a selection can be made in order to fit the developers needs. Nevertheless, only a small subset is necessary to implement fully functional CANopen devices.

The SDO service can be used as both server SDO (SSDO) and (CSDO). It is possible to define up to 128 of each kind. The SDOs use the expedited transfer for data up to 4 bytes. For data larger than four bytes segmented transfer is used. With this service, data up to 127 bytes, and, if they are marked as domain, up to 2^{32-1} bytes, can be exchanged. Furthermore a program upload and download to a device is possible.
For large data transfer the faster Block Mode[2] was implemented. It can be used with or without CRC checksum polynomial, calculated on the run or per table and with variable block size. Additionally a fallback to the segmented transfer is implemented.

The user can define up to 512 Transmit PDOs (TPDO) and 512 Receive PDOs (RPDO) in each device. Dynamic PDO mapping is possible and can be done bit-wise. Furthermore the dummy mapping can be used. The PDOs are usable with all transmission types. Cyclic and acyclic synchronous and asynchronous PDO are implemented. It is possible to change the transmission type during run time. Additionally transmit PDOs can be sent timer driven. To save PDO identifier resources Multiplexed PDOs can be used in all defined modes (Source Address Mode (SAM) and Destination Address Mode (DAM)). As a further feature it supports remote requests (RTR) also for Basic-CAN controllers.[3]

The EMCY services are available for both producer and consumer. One producer EMCY and an "unlimited" number of consumer EMCYs can be defined. Additionally the CANopen Library handles the error stack management (1003_h).

For SYNC and TIME services one producer and one consumer communication object per device is possible. An exception are devices with multiple CAN lines. They support one server and one client per line.

The Node Guarding protocol is implemented for both master and slave. Each of them checks the guarding time. If the guard time is elapsed, the application receives this information. The guarded slave can monitor if it is guarded within the expected time interval by a Node Guarding master.

Heartbeat for producer and consumer is available. If the Heartbeat producer Time is set (1017_h), Heartbeat starts immediately. If the Heartbeat consumers are active the application receives information when the boot-up message is received, an error occurs, or Heartbeat is restarted.

Another aspect during the development of the CANopen Library was the convenience for the users. An easy interface for the usage of all services has been implemented. Furthermore checking functions for parameter limits, data sizes and access attributes are available. The communication behavior in respect to CiA-301 is done fully by the CANopen Library i.e. default COB-ID distribution. The user has only to define his specific

[2] CiA-301, V4.02 chapter 9.2.2.1.8 and 9.2.2.1.12

[3] Please refer to the document CiA-AN802: "CANopen statement on the use of RTR-messages" for use of RTR messages.

application behavior on certain communication events.

For devices which own a nonvolatile memory the save and restore parameter handling can be used.

The further criterion is the security. Many features have been included in the CANopen Library in order to build robust applications.

The application is informed about any CAN errors. Values can be checked before they are written to the object dictionary. Additionally a resource security mechanism for multi-tasking systems and an interface for enabling/disabling of interrupts has been prepared.

The last criterion is the independence of the CANopen Library of the underlying hardware and operating system. It is easy to adapt the prepared driver modules to any target system. A further aim is to support a wide range of standard hardware products like CAN cards for PCs or micro controller modules.

3.2. Design Flow

The CANopen Library is only one component in the design flow of the CANopen system development. The development of CANopen systems consists of the two tasks, implementation and integration. In the context of the CANopen Library the implementation task must always be done.

Implementation Tasks

- Implementation of CANopen application
- Test of CANopen application
- Integration into the target system (hardware / operating system)
- Test of the whole device

Integration Tasks

- Integration of devices into a network
- Configuration of devices
- Test of communication behavior for distributed application
- Downloading of configuration data to control application

port provides a tool-based design flow for implementation and integration. The *CANopen Design Tool* by *port* supports the implementation part (figure 9).

It is a tool for creating and editing of CANopen device databases. These databases contain information about the device, which describes the interface to the CANopen network. These are in general the parameter, data, control and status information of the device accessible via CAN. These values are organized in the object dictionary. The device databases can be created from ready to use databases, which contain data of the standardized CANopen device profiles. The tasks of the *CANopen Design Tool* are configuration of the CANopen Library, managing the device data in a database and generating an object dictionary implementation (C-code), an Electronic Data Sheet (EDS) and a documentation of the implemented objects. This tool takes over error-prone tasks and prevents repetitious jobs. It consequently relieves the job of the developer.

The generated C-source code of the object dictionary is included by the application modules. This ensures the direct access to the variables (via variable names) and the access

via index and sub-index. The object dictionary is the data interface between the CANopen Library and the application.

A further result of the tool is an Electronic Data Sheet (EDS) according to CiA-306. The EDS should be delivered with every CANopen device. The EDS contains all relevant information about the device. This information is used by configuration tools and control applications in order to integrate the device in a network.

The *CANopen Design Tool* also generates a documentation of the implemented objects. Every parameter of the device is described in a table. Additionally a short descriptive text provides information about the object content and usage. At the start of each CANopen device development this documentation can be used as a part of the device specification. Later it can be included in the user and sales documents[4].

An enhancement to this are templates. These are source code skeletons containing behavior descriptions.

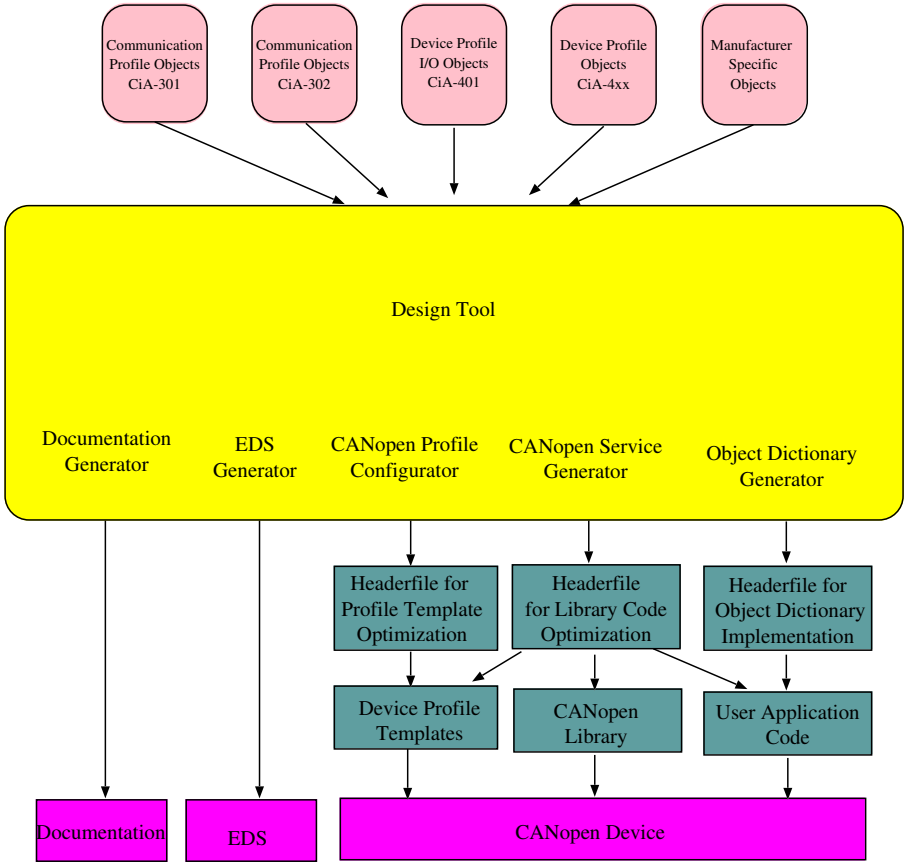


Figure 9, Implementation Design Flow

If the target system is not yet available i.e. hardware is still under development, a cross development can be done. For that purpose a host development environment, using Windows™ or Linux has been developed. Under this system the whole application can be written without any hardware dependence. If no Linux system is available a common

[4] For detailed information see User Manual *CANopen Design Tool*.

standard Windows ™ PC with a PC-CAN card can be used[5].

After the cross development, the application must be integrated into the target system. This is done by changing the CANopen Library driver modules. In the application code modules **no** changes are necessary provided the hardware parts are coded using a hardware abstraction layer (HAL).

Commercial configuration tools and CAN analyzers can be used for the device tests.

The next step is the device integration into a network. A distributed application is created using this network. It is very convenient to use a configuration tool. This tool loads the EDS-files of the participants (devices). Then the communication channels can be built by distribution of COB-IDs and PDO mapping. Further the application parameters can be changed. Afterwards a so-called Device Configuration File (DCF) is stored. The configuration data can be downloaded onto a control application, which configures the network after each boot-up. Therefore the configuration tool is not longer necessary within the network. It fulfills only maintenance requirements.

A summary of the complete tool-set is shown in figure 10.

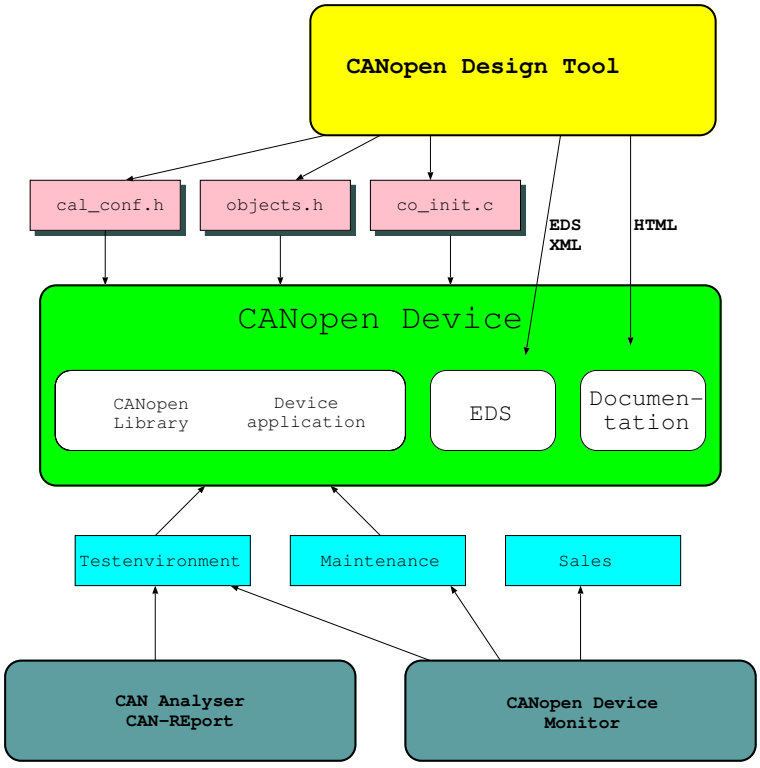


Figure 10, CANopen Tool Set

[5] The *CANopen LINUX Starter Kit* can be offered for that purpose.

3.3. CANopen Library Structure

The structure of the CANopen Library is shown in figure 11. The CANopen Library consists of a hardware-dependent and an hardware-independent section. The two sections are coupled together by message buffers (CAN receive and CAN transmit buffer).

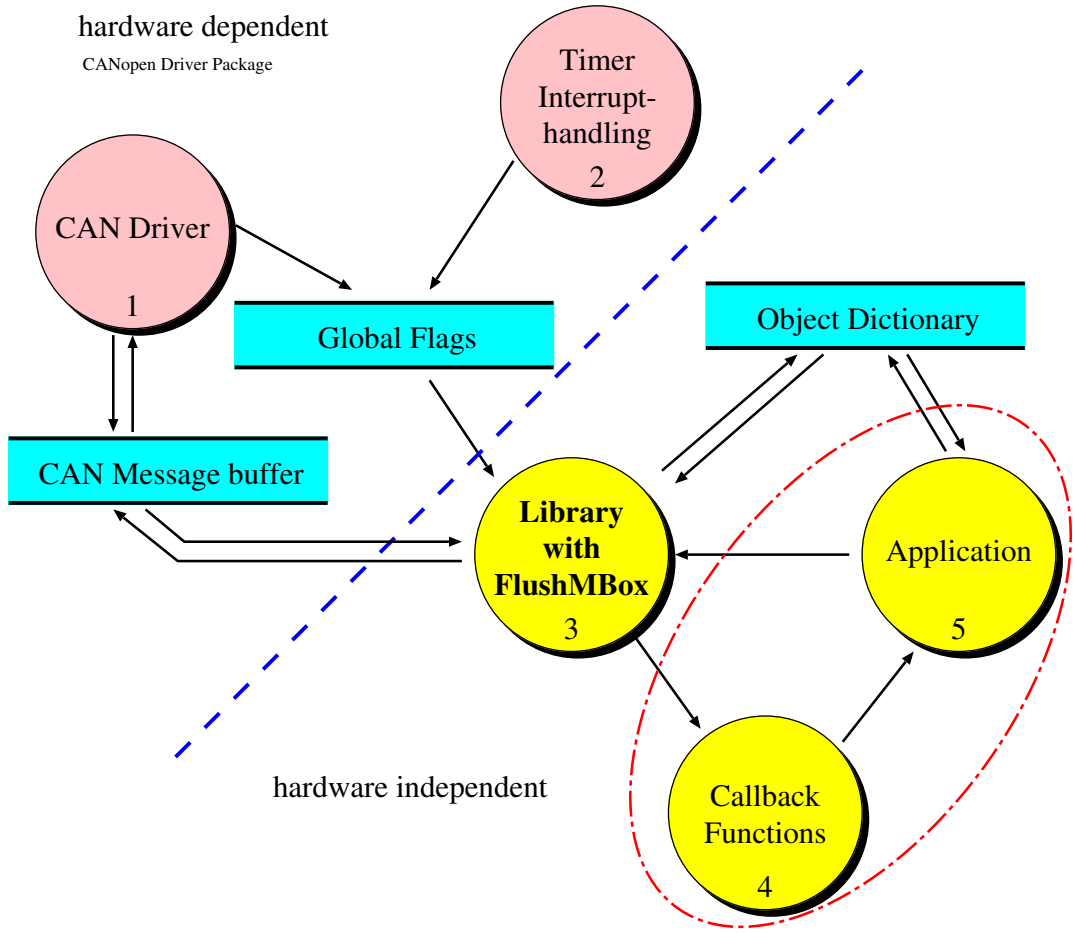


Figure 11, CANopen Library Data Flow

The hardware dependent section consists of the components:

CAN Driver

- operation of the CAN controller (interrupt-controlled)
- receipt of CAN Messages and entry in CAN receive buffer
- reading the transmit buffer and transmitting the CAN telegrams (interrupt-controlled)
- monitoring the CAN controller (set error flags on errors)

Timer Interrupt

For all timing related tasks like Heartbeat producing and consuming, PDO inhibit time monitoring and others.

- supply of a defined timer interval
- setting appropriate timer flags

The hardware-independent section consists of the:

- CANopen Library with FlushMbox
  - evaluation of the error flags of the CAN controller
  - evaluation of the timer flag and call of the timer-dependent services (SYNC, Node Guarding, Heartbeat)
  - calling of appropriate CANopen service routines depending on contents of messages received by CAN receive buffer
    possibly calling callback functions of the CANopen Library containing application code
  - updating of object directory entries with new data

Callback Functions

- called from the CANopen Library - application-specific reaction for CANopen services
- are to be filled out by the user

User Application

- application behavior
- initializing CANopen services
- call CANopen service requests
- update object directory entries with user data

The structure of the CANopen Library is reflected in the file structure of the program components too.

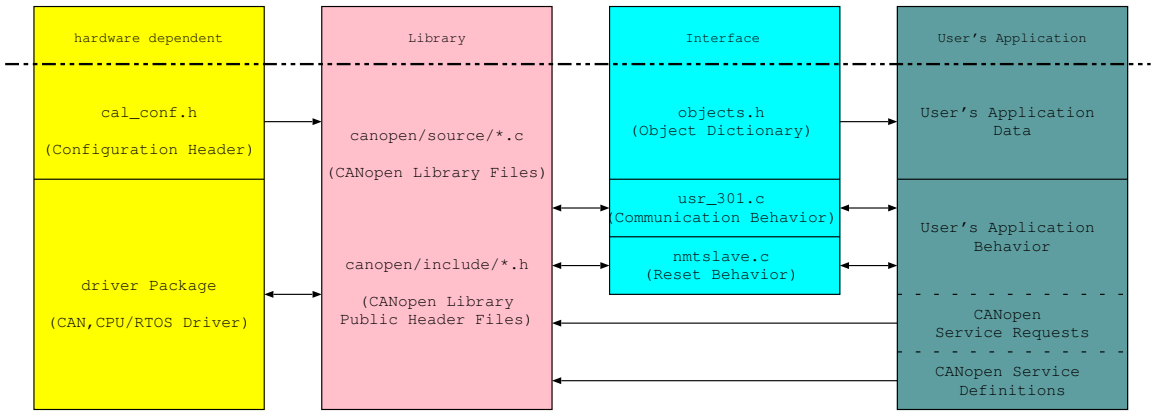


Figure 12, The File Structure of the CANopen Library

The configuration of the CANopen Library (device type, services, communication mechanism, etc.) and the hardware characteristics such as CAN controller type, interrupt, operating system, is written to the header file **cal_conf.h**. It is stored in the application directory because it contains configuration data for all CANopen components of an application.
All hardware drivers (**cpu.c**, **can.c**) for the access to the CAN controller and the timer handling are stored in the subdirectory *drivers*.

The interface between the CANopen Library and the application consists of the callback functions and the object dictionary. The callback functions are supplied in the files **usr_301.c**, **usr_302.c**, **usr_303.c**, **usr_30x.c**..., and **nmtslave.c** and have to be filled out application-specific by the application programmer. The object dictionary contains data which are either application-specific or CANopen-specific. These interface functions belong to the application and are stored in the application directory. All CANopen Library files are stored in the directory path *canopen/source*.

Thus also an update of the entire CANopen Library without modification of the application-specific files is possible.

3.3.1. Object Dictionary

The object dictionary is the data interface between the user's application and the CANopen Library. It contains all variables that are necessary for the CANopen services and the application-specific variables that should be accessible over the network.
The implementation of the object dictionary by *port* consists of an array of element headers for each used index (figure 13).

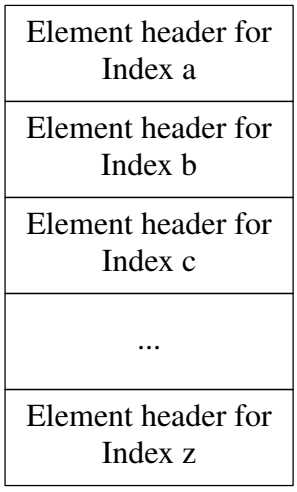


Figure 13, Structure of the Object Dictionary Array

Each element header contains five entries:

- object dictionary index number

- number of elements for this object dictionary entry (number of sub-indexes)
- pointer to the real variable data
- pointer to the description structure for the object
- pointer to callback function for the object

The real object can be a simple variable, an array, record or domain entry and can be created with the object dictionary or by the user application (figure 14).

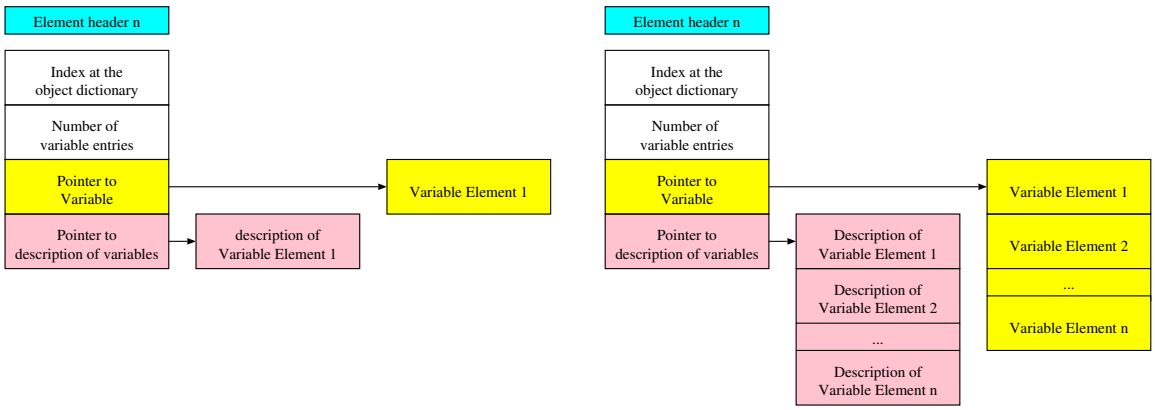


Figure 14, Object Dictionary Implementation

The description for each variable is an C-language record. It contains entries for the value ranges, the default value, the size, read/write permission flags, numeric and domain identification and PDO mapping permission.

With the multi CAN line version an object dictionary is available at each line. In this case all object dictionaries will be managed by an object dictionary manager. This manager is an array of pointers to the implemented object dictionaries (figure 15).

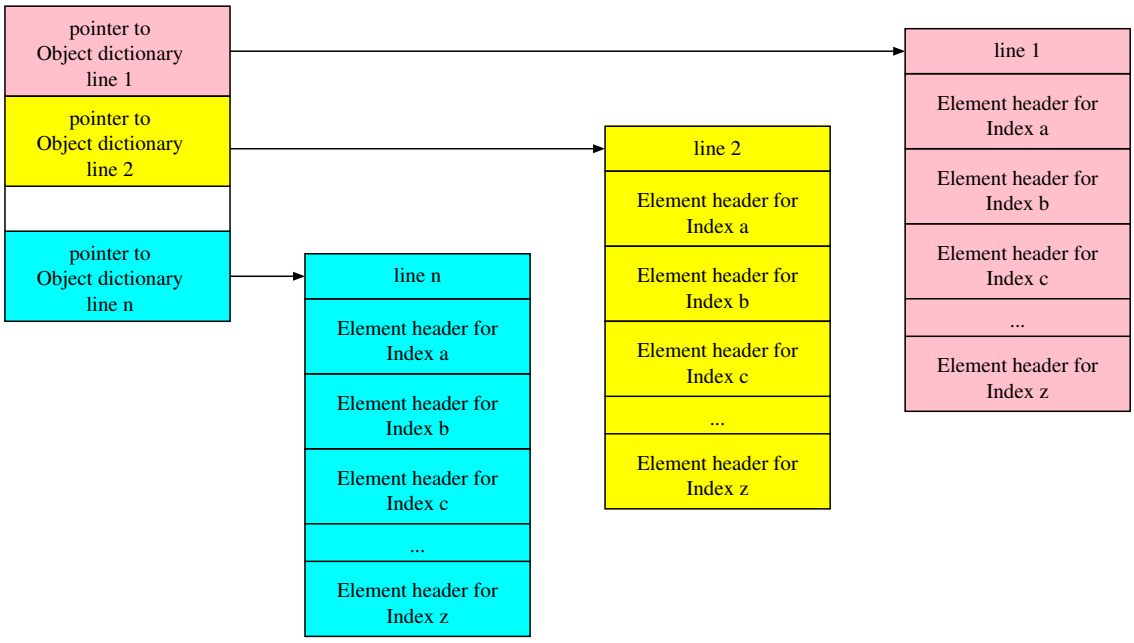


Figure 15, Structure of Multi-Line Object Dictionary

It is possible to define an interface for one's own variables, structures and arrays. The index is the logical reference to one of these data containers. The elements of structures and arrays are reached via the sub-index. For simple variables the sub-index is always 0. If the size of the structures or arrays is bigger than 255 bytes (limit of sub-index), the user must split them. This means more than one index is needed to describe the structure/array within the object dictionary. The type `LIST_ELEMENT_T` is the element header. An array of this type, sorted by the index, is the object dictionary. The description of the variables is an array of the type `VALUE_DESC_T`. Each array entry describes the properties of the corresponding sub-index.

```
typedef struct
{
    UNSIGNED8    *pObj;
    VALUE_DESC_T *pValDesc;
    UNSIGNED16   index;
    UNSIGNED8    numOfElem;
#ifdef CO_CONFIG_ENABLE_OBJ_CALLBACK
    CO_OBJ_CB_T  pObjCallback;  /* obj function pointer */
#endif /* CO_CONFIG_ENABLE_OBJ_CALLBACK */

} LIST_ELEMENT_T;
```

Name	Description
index	index of the user's variable in the object dictionary
numOfElem	number of elements of user's variable
pObj	pointer to user's variable (array, structure, variable)
pValDesc	pointer to an array of corresponding value descriptions
pObjCallback	pointer to an callback function corresponding to the object

Table 14, Element Header Description

```
typedef struct
{
    UNSIGNED8   *pDefaultVal;
#ifdef CONFIG_LIMITS_CHECK
    LIMIT_U8_T  *pLimits;
#endif
    UNSIGNED8   varType;
    UNSIGNED16  attribute;

} VALUE_DESC_T;
```

Name	Description
pDefaultVal	pointer to the default value of the variable, it will be used to initialize the variable after a reset (hard reset or communication reset) it depends on the variable type (see varType)
pLimits	pointer to the lower and upper limits of the variable value it depends on the variable type (see varType)
varType	type of variable (table below)
attribute	attributes of value (table below)

Table 15, Variable's Properties Description

Bit	Property	Description (bit = 1)
CO_MAP_PERM	PDO Mapping Permission	PDO mapping for this entry is allowed
CO_WRITE_PERM	Write Permission	the variable is writable
CO_READ_PERM	Read Permission	the variable is readable
CO_CONST_PERM	Const value	this entry is constant and stored at ROM
CO_SHORT_ARRAY_DESC	Short description	the description for all sub-indices are in the sub-index 1, all following descriptions can be canceled
CO_OBJ_ATTR_SAVE	Non volatile storage marker	If this attribute is set, the object shall be stored in non volatile memory.

Table 16, Sub-Index Attributes Description

Bit	Variable Type
CO_TYPEDESC_BOOL	BOOL
CO_TYPEDESC_UNSIGNED8	unsigned char (8 bit)
CO_TYPEDESC_UNSIGNED16	unsigned int (16 bit)
CO_TYPEDESC_UNSIGNED32	unsigned long (32 bit)
CO_TYPEDESC_UNSIGNED64	unsigned long long (64 bit)
CO_TYPEDESC_INTEGER8	signed char (8 bit)
CO_TYPEDESC_INTEGER16	signed int (16 bit)
CO_TYPEDESC_INTEGER32	signed long (32 bit)
CO_TYPEDESC_INTEGER64	signed long long (64 bit)
CO_TYPEDESC_VISSTRING	visible string
CO_TYPEDESC_OCTETSTRING	octet string
CO_TYPEDESC_DOMAIN	domain
CO_TYPEDESC_REAL32	real (32 bit)

Table 17, Variable Type Description

The pointers to the default value and to the limit structure are always pointers to the real data type of the variable and must be casted at VALUE_DESC_T type.

The CANopen Library does not interpret float values except if limit check is enabled. Please ensure that the initialization values are in the right order.

3.3.2. CANopen Library Configuration

3.3.2.1. Configuration Header

The CANopen Library can be used for many different hardware and compiler platforms. Furthermore the CANopen Library can be configured to reduce code size and run faster. Due to the complexity of this process, the interactive is available for both Microsoft Windows and UNIX-machines® to support the creation of the configuration file **cal_conf.h**.

The entries of the file **cal_conf.h** determine the kind of compilation. All configuration compiler-define-directives used in that file have the prefix `CONFIG_`.

The user can compile the CANopen Library code for a CANopen network master or for a slave application.

For the multi-line version the number of CAN lines can be configured. Additionally the multi-line functionality can be switched off. This means all functions do not use the line select function parameter *canLine*. In this way the CANopen Library can be used for single line systems additionally without the multi line overhead.

Furthermore it is possible to reduce the code size by selecting parts of the code by compiler `#define` directives. Every CANopen service can additionally be separated into client/consumer or into server/producer functionality. If the user needs only one functionality, he only has to define one of these e.g. `CONFIG_SDO_SERVER`.
Furthermore the size of the CAN message buffer can be adjusted and the usage of Full-CAN properties in hardware can be enabled, if the CAN controller is a Full-CAN type.

For compilers (processors) which do not support a byte alignment of data in the memory, an alignment definition `CONFIG_ALIGNMENT` has to be set in order to ensure a correct access to structures and arrays of the object dictionary.

➥ For a detailed description of the compiler `#defines` see Appendix 4.

Please never edit the **cal_conf.h** by hand !

3.3.2.2. Coding of 64-bit Values

Some entries at the object dictionary are of type `UNSIGNED64`. At the moment, not all compilers provide this data type. In this case a special type can be used.

```
typedef struct {
    charval[8];
} UNSIGNED64;
```

The initialization in this case is done by the following macro:

```
#define SET_U64(b1, b2, b3, b4, b5, b6, b7, b8)    \
    { b8, b7, b6, b5, b4, b3, b2, b1 };
```

For all compilers providing this data type the initialization is done by:

```
#define SET_U64(b1, b2, b3, b4, b5, b6, b7, b8)    \
    { b1<<56|b2<<48|b3<<40|b4<<32|b5<<24|b6<<16|b7<<8|b8 };
```

4. Using the CANopen Library

4.1. Service Definition Interface

All CANopen services the application needs must be defined before they can be used. Therefore definition functions are available for each service. These definition functions setup the service parameters to their default values.

The name of the functions for the definition interface always starts with the prefix "*define*" followed by an abbreviation of the service object e.g. *definePdo*().

If the function returns the value "**success**" (return value: `CO_OK`), then the created service can be used by the application.

The picture below illustrates the naming scheme for functions that use CANopen services.

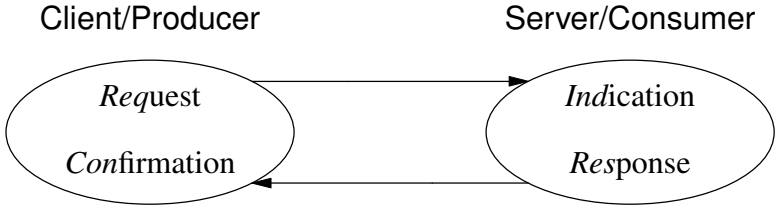


Figure 16, Naming Scheme for Functions

4.2. Service Request Interface

In order to be able to use CANopen service functions, request functions are available. The function name consists of three sections. The first word of the service request name is the abbreviation of the CANopen service object (*read*, *write*, *start* ..). The second word is the kind of service (PDO, SDO, SYNC..) and finally the abbreviation of the word request "*Req*", e.g. *writePdoReq*().

There are two kinds of request functions. The first kind of requests directly execute the request in the user application. These are all the requests, which can be completely executed immediately by the CANopen layer, e.g. *startSyncReq(), writePdoReq(),...).*
The second kind of requests result in a response or confirmation from another node in the network. Here the request can only be instructed e.g. *readSdoReq(), readPdoReq(),...* because an interaction with another node in the network is necessary. The application can determine the completion of the request, e.g. the reception of a response, using the confirmation functions.

Error free return from the request functions (`CO_OK`) means that the statement was executed error free up to putting created CAN messages into the CAN transmit buffer. The successful transmission of a CAN message is the job of the CAN driver (not of the request function) and depends, among other things, also on the current bus load.

4.3. Service Indication/Confirmation Interface

With the reception of certain CAN messages, error conditions (e.g. Heartbeat message is missing, Timeout occurred) and other events (e.g. completed request) the user is informed by the CANopen Library by indication or confirmation functions also referred to as call-back -functions.
The confirmation is an answer to a confirmed service request (e.g. SDO). All other events are so-called indications. Function names that are CANopen service indications and confirmations are appended with "*Ind*" e.g. *pdoInd*() as the abbreviation for indication and "*Con*" e.g. *sdoRdCon*() as abbreviation for confirmation. The function prefixes of the indication and confirmation have the same meaning as in the request interface description.

The receive principle for the service indication/confirmation is shown in figure 17.

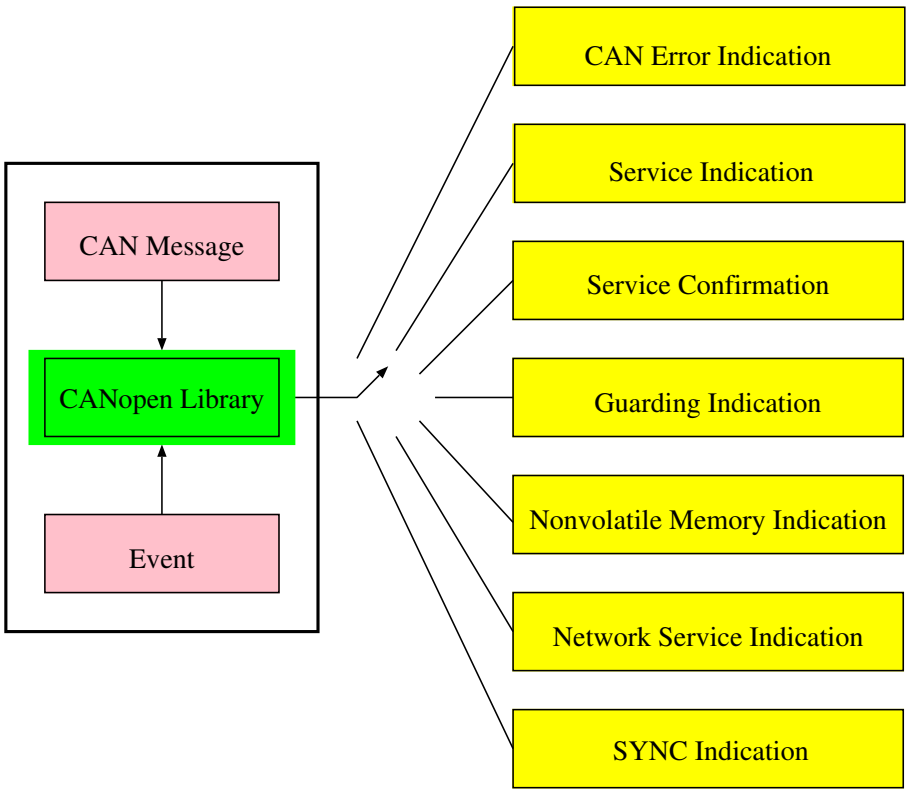


Figure 17, The Indication/Confirmation Interface

It is possible to define an error handling for interrupted SDO transfer (Abort-Domain-Transfer-Service), for guarding indications (lost guarding, start/failed Heartbeat, boot-up message), for an emergency message reception or for errors received from the CAN controller (driver).

All user interface functions have been defined in the modules **usr_301.c**, **usr_302.c**, **usr_303.c**, **usr_304.c**, **usr_305.c** and **nmtslave.c**. The file names are derived from the corresponding CANopen standards, e.g. **usr_301.c** relates to CiA-301. However, only

the function calls are defined in the provided template modules. The behavior of the indication or confirmation functions must be determined (means "coded") by the application programmer.

Function	Kind of Service	Contents (user reaction)
getNodeId()	Local Initialization	node-ID read function
pdoInd()	Service Indication	PDO indication
mpdoInd()	Service Indication	MPDO indication
emcyInd()	Service Indication	emergency indication
timeInd()	Service Indication	time stamp indication
testSdoValue()	Service Indication	value test before writing
sdoRdInd()	Service Indication	SDO read indication
sdoWrInd()	Service Indication	SDO write indication
sdoRdCon()	Service Confirmation	SDO read confirmation
sdoWrCon()	Service Confirmation	SDO write confirmation
mGuardErrorInd()	Guarding Indication	guarding handling on master
sGuardErrorInd()	Guarding Indication	guarding handling on slave
clearParameterInd()	Nonvolatile Memory	clear memory command
loadParameterInd()	Nonvolatile Memory	load memory command
saveParameterInd()	Nonvolatile Memory	save memory command
canErrorInd()	CAN Error Indication	CAN error handling
syncPreCommand	SYNC Indication	before SYNC process
syncCommand	SYNC Indication	after SYNC process
lssMasterCon()	LSS Confirmation	confirmation
lssSlaveInd()	LSS Indication	indication
ledInd()	LED Indication	Set/Reset LED request

Table 18, Some User Receive Interface Routines of the module **usr_30x.c**

If a function listed above is called a message was received or an event had occurred. The corresponding values in the object dictionary had been updated before the call. Some of the functions require certain conditions to be met. These conditions are described in the following chapters.

4.4. Configuration Interface

Most of the CANopen service functions use predefined COB-identifiers for communication. These predefined COB-identifiers are determined on the basis of the own node-ID. For high flexibility the CANopen Library uses a function to determine the node-ID.
Within the function *getNodeId*() the user determines the node-ID, e.g. by reading out some DIP switches. The CANopen Library calls this function once from *initCANopen*() and once from the *resetCommInd*() function.

4.5. Timer Usage

The CANopen Library uses an internal timer concept that can be used for application specific purposes, too. As a basis a hardware timer is used. It is included by the driver and increments a variable in an predefined interval. This interval is called *timer tick* and is the smallest resolution of timer dependent processes. All timer dependent processes of the CANopen Library can only be executed in multiples of *timer ticks*. *Timer ticks* are counted normally in an UNSIGNED16 variable. If the define `CONFIG_LARGE_TIMER` is set, timer ticks are counted in an UNSIGNED32 variable. Therefore the maximum value of a timer event is 0xffff * length or 0xffffffff * length of a timer interval.

The timer itself does not need any additional memory. Therefore any desired number of timer processes can be started. For every function that needs a timer a static timer structure has to be provided of the calling function. All timer structures are administered in a linked and sorted list. This makes it possible that even with many timers there is no loss in execution time. After the time has run out the indication function *userTimerEvent()* is called, in which the user can specify further actions. Furthermore it is possible to use the timer as a cyclic timer. The following functions provide the programming interface to the timer.

Name	Function
addTimerEvent()	add a timer event to the timer list
removeTimerEvent()	delete a timer event from the timer list
changeTimerEvent()	modify an active timer event
checkActiveTimer()	check for an active timer
userTimerEvent()	user indication - timer has been finished

Table 19, Timer Functions

By using the function *addTimerEvent*() the new timer is added. When the timer is elapsed the indication function *userTimerEvent*() is called. In this function the user can specify further actions. When the timer has to be switched off before time is up *removeTimerEvent*() can be called. All timer functions expect as first parameter a pointer to the data structure of the timer. This structure has to be provided as static data from the calling function. The structures are modified by the timer functions. Therefore the user program must not alter the data of the static timer structures.

The second parameter specified the timer interval in 1/10 of msec. And the third parameter is the timer type. For application specific timers it should be set to *CO_TIMER_TYPE_USERSPEC*. For cyclic timers additional the attribute *CO_TIMER_TYPE_CYCLIC* has to be set.

Example of the usage of a timer:

```
TIMER_EVENT_T   myTimer;        /* define timer struct */

/* add a cyclic timer for 1 sec */
addTimerEvent(&myTimer, 10000, CO_TIMER_TYPE_USERSPEC | CO_TIMER_TYPE_CYCLIC);

 ...

void userTimerEvent(TIMER_EVENT_T *pTimer) {
    if (pTimer == &myTimer)  {
        /* start my reaction */
    }
}
```

Listing 1, Example for Timer Usage

4.6. SDO Usage

SDO transfers are always peer-to-peer connections between two nodes - a server node and a client node. The client is using SDO read or write requests to access the servers object dictionary. SDO transfers are confirmed services and therefore 2 COB-IDs are necessary for each connection, one for the request, one for the response. Each node can have many SDO connections and it can be a server, client or both.

There are three different transfer modes possible: Expedited Transfer, Segmented Transfer and Block Transfer. The CANopen Library selects automatically the best mode for each transfer.

If the node permits the access to its own object dictionary, it must provide at least one server SDO connection, i.e. by creating an SDO communication object using the *defineSdo*() function call, or more than one if more clients should have access to its object dictionary. If the node wants to access the object dictionary of other nodes it has to initialize a

for each node it wants to connect to, also using the *defineSdo*() function call.

All SDO communication services have to be initialized by the function *defineSdo*() (Listing 2).
Except the first server SDO, all SDOs are marked as invalid after the initialization. Only the COB-IDs for the first server SDO are initialized with the default COB-ID on the basis of the node-ID (see pre-defined connection set). The COB-IDs for the other SDOs should be set and validated using the function *setCobId*() (Listing 2).

```
defineSdo(1, SERVER);                   // set the COB-ID not necessary
defineSdo(1, CLIENT);                   // define SDO as client SDO

cobId = 1200;                           // define COB-ID
setCobId(0x1280, 1, cobId);             // validate cob-id client-server

cobId = 1201;                           // new COB-ID
setCobId(0x1280, 2, cobId);             // validate cob-id server-client
```

Listing 2, Example for Defining SDOs and Setup COB-IDs

4.6.1. SDO-Server

The SDO server permits access to the own object dictionary to other nodes via the CANopen network. For the access to the three mandatory objects in the object dictionary every CANopen node must have at least one server SDO object. Only the first server SDO is available immediately after the initialization. If more server SDOs should be used the COB-IDs for those SDOs have to be configured (Listing 2).

All attempts of a read or write access from a remote node to the own object dictionary are indicated by the *sdoRdInd*() and *sdoWrInd*() functions.

A read access from any other node in the network to the own object dictionary is indicated by the function *sdoRdInd*(). In this function the application can update the requested value (the object dictionary entry addressed by **index** and **sub-index**) before the CANopen Library sends back the response message to the client. If the indication function *sdoRdInd*() returns an error, an SDO abort transfer will be generated and is sent back to the originator (the read service requester).

Error codes are generated automatically by the CANopen Library, see appendix 5.

```
/*****************************************************************
*
* sdoRdInd - indicates the occurrence of an SDO read access
*
* \retval CO_OK success
* \retval CO_E_xxx error
*
*/

RET_T sdoRdInd(
          UNSIGNED16 index,       /* index to object */
          UNSIGNED8  subIndex     /* index to object */
#ifdef CONFIG_MULT_LINES
         ,UNSIGNED8 canLine      /* number of CAN line 0..CONFIG_MULT_LINES-1 */
#endif
    )
{
    /* increment the counter before send back the value */
    actual_u32++;
    return(CO_OK);
}
```

Listing 3, Example s1, SDO Read Indication

A write access to the own object dictionary is indicated by the function *sdoWrInd*() (figure 18).

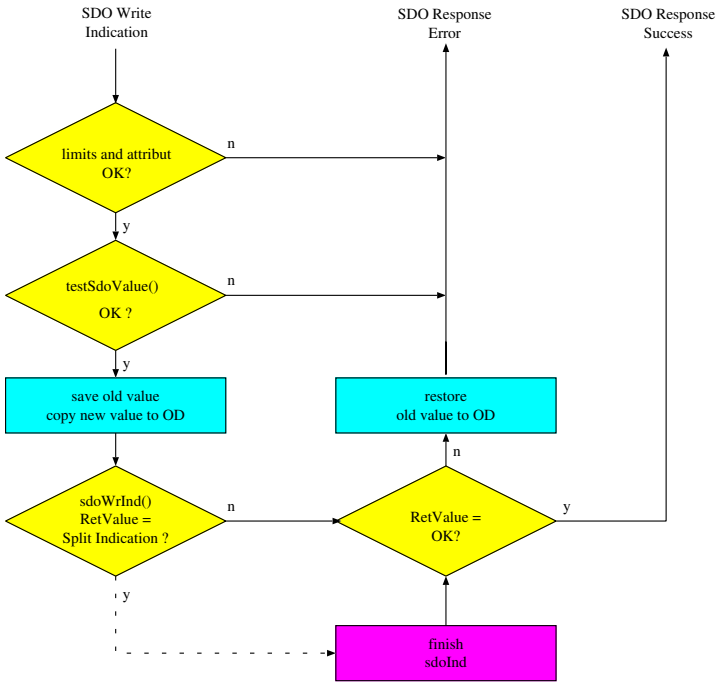


Figure 18, The SDO Write Indication Flow

First the write permission flag and the value limits are tested by the CANopen Library. If the value is within the limit range the user can test the value by the function

testSdoValue() before it is written into the object dictionary. This test can be necessary if the application uses the corresponding variable via a second task or interrupt service routine or for data with size greater than 4 bytes. For variables with size less than 4 bytes (e.g. UNSIGNED8-UNSIGNED32, INTEGER8-INTEGER32, REAL32 values) the old value is stored before the new value is written into the object dictionary and the function *sdoWriteInd()* is called. In the case of an error return value from *sdoWriteInd()* the old value is restored.

```
/*****************************************************************
*
* sdoWrInd - indicates the occurrence of an SDO write access
*
* \retval CO_OK success
* \retval CO_E_xxx error
*
*/

RET_T sdoWrInd(
          UNSIGNED16 index,       /* index to object */
          UNSIGNED8  subIndex     /* sub-index to object */
#ifdef CONFIG_MULT_LINES
         ,UNSIGNED8 canLine      /* number of CAN line 0..CONFIG_MULT_LINES-1 */
#endif
    )
{
    actual_u32++;
    /* Look if index 0x2000 = setpoint */
    if ((index == 0x2000) && (subIndex == 0)) {
        actual_u32 = setpoint_u32;
    }
    return(CO_OK);
}
```

Listing 4, Example s1, SDO Write Indication

4.6.2. SDO-Client

The SDO client initiates all SDO transfers. For each SDO client connection an SDO communication object has to be initialized. After the initialization all s are disabled by default. To enable these SDOs the COB-IDs must be set according to the server COB-IDs - which should be contacted - with *setCobId*() (Listing 2).

Read and write access to the object dictionary of another node is started with *readSdoReq*() and *writeSdoReq*(). Parameters for the function calls are the SDO number, index and sub-index in the object dictionary of the SDO server and a data buffer for transmission data.
The application has to ensure that the data buffer is large enough for all data which are to be transferred.

An error free return value from the request functions does not necessarily mean a successful transmission. It means only that the transfer is initiated by saving the first data

into the transmit message buffer.

The application is informed about the termination of the transfer through the functions *sdoWriteCon*() and *sdoReadCon*(). If an error occurs these functions can evaluate the error reason.

```
void sdoWrCon(
       UNSIGNED8  sdoNr,        /* number of SDO */
       UNSIGNED32 errorFlag     /* errorflag, if zero success */
#ifdef CONFIG_MULT_LINES
       ,UNSIGNED8 canLine       /* number of CAN line 0..CONFIG_MULT_LINES-1 */
#endif
    )
{
    if (errorFlag == E_SDO_TIMEOUT) {
        printf("Timeout");
        return;
    }

    switch (errorFlag & 0xFF000000UL) {
        /* successful confirmation */
        case E_SDO_NO_ERROR:
            break;
        /* service error */
        case E_SDO_SERVICE:
            printf("Error: Service Error");
            switch(errorFlag & 0x00FF0000UL )
            {
                case E_SDO_INCONS_PARA:
                    printf(" - Inconsistent parameter");
                    break;
                /* wrong communication parameter */
                case E_SDO_ILLEG_PARA:
                    printf(" - Illegal parameter");
                    break;
            }
            break;

        ...

        default:
            printf("Error: abort dom transfer reason %lX", errorFlag);
                break;
    }
}
```

Listing 5, SDO Write Confirmation

If an SDO server does not respond to the request from the SDO client within a determined period of time the SDO client can abort the transfer with an **Abort Domain Transfer Protocol**. This is done automatically by the CANopen Library when the time, given by *writeSdoReq*() or *readSdoReq*() is up. The application is informed by the indication function *sdoRdCon*() or *sdoWrCon*() about this event.

SDO communication up to four bytes can be transmitted by so-called Expedited Transfer. This means, all data can be passed in one CAN telegram. For larger data the segmented transfer has to be used. The CANopen Library forces automatically the transfer type by the requested byte count.

4.6.3. Domain Up/Download

A domain in CANopen is unstructured data, which can have a size up to 2^{32} bytes. Domains can be whole application programs or large data structures, e.g. pictures. The application is always responsible for the interpretation of the domain content.

Domains can only be transferred by SDO. In order to handle such large data some exceptions to the common CANopen objects are necessary. All objects with domain entries have the type `DOMAIN_T`. This type is a pointer to `void` and is initialized with `NULL` in **objects.h** generated by the *CANopen Design Tool*. The data type `DOMAIN_T` is a basic data type and can be used as variable, in arrays and in records.

```
DOMAIN_T        man_domain_var = { NULL };
DOMAIN_FIELD2_T man_domain_array = { 0x2, { NULL, NULL }};

DOMAIN_DATA_T   domain_data[] = { 0, 0, 0 };
UNSIGNED8       defaultVal_U8 = { 2 };

VALUE_DESC_T    man_updown_domain_desc[1] = {
                    {   (UNSIGNED8 *)&domain_data[0],
                        CO_TYPEDESC_DOMAIN,
                        CO_READ_PERM | CO_WRITE_PERM }
                };
VALUE_DESC_T    test_desc[3] = {
                    {   &defaultVal_U8[0],
                        CO_TYPEDESC_UNSIGNED8
                        CO_READ_PERM | CO_WRITE_PERM | CO_NUM_VAL },
                    {   (UNSIGNED8 *)&domain_data[1],
                        CO_TYPEDESC_DOMAIN,
                        CO_READ_PERM | CO_WRITE_PERM },
                    {   (UNSIGNED8 *)&domain_data[2],
                        CO_TYPEDESC_DOMAIN,
                        CO_READ_PERM | CO_WRITE_PERM }
                };
```

Listing 6, Domain Declaration in *objects.h*

The application programmer is responsible for setting the pointer the correct data area and the size of the domain objects.

```
/* users module */

/* area for domain storage */
UNSIGNED8 programDownloadArea[MAX_DOMAIN_BUF_SIZE];

main()
{
    /* assign address space to domain from objects.h with
     * index DOMAIN_INDEX and sub-index DOMAIN_SUB */

    setDomainAddr(DOMAIN_INDEX, DOMAIN_SUB, &programDownloadArea[0]);
}
```

Listing 7, Initialization of a Domain

In order to manage remote data area from a node, some functions have been introduced for manipulating the start address and the domain size. This is useful for building ring buffers and other segmented buffer structures e.g. for drive interpolation data. Furthermore certain segments of a domain can be uploaded e.g. for program debugging.
The manipulation functions are listed below:

- *setDomainAddr*()
- *getDomainAddr*()
- *setDomainSize*()
- *getDomainSize*()

Listing 8, Functions for Manipulating Domain Variables

Domain transfers can be started by normal SDO functions *writeSdoReq*() or *readSdoReq*(). After the transfer is finished, the normal indication functions *sdoRdInd*() and *sdoWrInd*() will be called, respectively. In some applications an indication function after a defined block of transferred data is necessary, because the receive buffer is not large enough or the data should be flashed into a ROM area. The CANopen Library can handle this for the SDO client and the SDO server for upload and download transfers.

All SDO transfers are initialized by the SDO client, so the SDO server is always the passive part. Therefore the indication size for the SDO server must be setup at compile time. It can be done by the *CANopen Design Tool*. If the configured data size is elapsed, the indication function *sdoDomainInd()* is called. Here the application can save or flash the received data. After that, the receive buffer will be cleared and the next data will be received until the next border of the configured data size is reached. Then the indication function is called again.

```
RET_T sdoDomainInd(
        UNSIGNED8       *pData,         /**< pointer the domain buffer */
        UNSIGNED32      actSize,        /**< number of Bytes to flash */
        UNSIGNED8       overSize        /**< number of Bytes to buffer */
    )
{
                                        /* temporary flash buffer */
UNSIGNED8 flashBuffer[CONFIG_DOMAIN_INDICATION_SIZE];
```

```
static UNSIGNED8 savedBuffer[7];        /* static save buffer */
static UNSIGNED8 savedBufferSize = 0;   /* count of saved data */

    /* first copy saved bytes to flash buffer */
    memcpy(&flashBuffer[0], &savedBuffer[0], savedBufferSize);

    /* now copy new received bytes to flash buffer */
    memcpy(&flashBuffer[savedBufferSize], pData, actSize);

    /* save oversize data for next flash cycle */
    memcpy(&savedBuffer[0], pData + actSize, overSize);
    savedBufferSize = overSize;

    /* flash data */
    return(CO_OK);
}
```

Listing 9, Example Code from *sdoDomainInd*() Function of an SDO server

The SDO client can define the size for the confirmation individually for each transfer by start a domain transfer using the function *writeSdoDomainReq()* or *readSdoDomainReq()*. If the given confirmation size is reached, the corresponding indication function *sdoDomainRdCon()* or *sdoDomainWrCon()* is called, before the next CAN message is transferred. At the end of the transfer, the normal indication functions *sdoRdCon()* or *sdoWrCon()* is called.

```
RET_T sdoDomainRdCon(
        UNSIGNED32      actSize,        /**< number of Bytes to flash */
        UNSIGNED8       overSize        /**< number of Bytes to buffer */
    )
{
                                        /* temporary flash buffer */
UNSIGNED8 flashBuffer[CONFIG_DOMAIN_INDICATION_SIZE];
static UNSIGNED8 savedBuffer[7];        /* static save buffer */
static UNSIGNED8 savedBufferSize = 0;   /* count of saved data */

    /* first copy saved bytes to flash buffer */
    memcpy(&flashBuffer[0], &savedBuffer[0], savedBufferSize);

    /* now copy new received bytes to flash buffer */
    memcpy(&flashBuffer[savedBufferSize], pData, actSize);

    /* save oversize data for next flash cycle */
    memcpy(&savedBuffer[0], pData + actSize, overSize);
    savedBufferSize = overSize;

    /* flash data */
    return(CO_OK);
}
```

Listing 10, Example Code from *sdoDomainRdCon*() Function of an SDO client

For program and firmware download CANopen defines the objects $1F50_h$ for the program download and the object $1F51_h$ for program control. For this the following code

snippets can be used.

Example code from *sdoWrInd*() function of an SDO-Server:

```
RET_T sdoWrInd(
          UNSIGNED16 index,     /**< index to object */
          UNSIGNED8  subIndex   /**< sub-index to object */
          )
{
unsigned char bank;

    switch(index) {
      case 0x1f50:

        /* program or configuration data download
         * subIndex does specify the Flash bank
         * that has to be programmed
         * !! subIndex differentiates between firmware
         * and config area
         */
        if ((subIndex > 0) && (subIndex < 5)) {
            /* Firmware */
            bank = subIndex + 7;
        } else if ((subIndex > 4) && (subIndex < 9)) {
            /* config data */
            bank = subIndex - 1;
        } else {
            return CO_E_NOT_EXIST;      /* failure, wrong sub-index */
        }

        /*         bank, source address */
        BDEBUG("Download to SubIndex %d, bank %d\n",
                             (int)subIndex, (int)bank);
        /* optional fill until buffer end with 0xff
         * in case there are not enough data in the domain
         */
        ret = program_flash(bank, GS_download_area);
        BDEBUG("Download ret with %d\n", (int)ret);
        man_last_flash_error = ret;
        memset(GS_download_area, 0xFF, MAX_UPLOADAREA);
        return ret;
        break;

      case 0x1f51:
        if ((subIndex == 1) && (p301_prog_control[1] == 1)) {
            firmware_prog();
        }
        if ((subIndex == 2) && (p301_prog_control[2] == 1)) {
            load_config();
        }
        break;
    }

    return CO_OK;
```

```
}
```

Listing 11, Example Code from *sdoWrInd*() Function of an SDO server

Example Code of an SDO client:

```
void *pBuffer;
RET_T commonRet;

fd_infile = open(cmdToken[3], O_RDONLY);
if (fd_infile < 0)  {
    fprintf(stderr, "can not open download file %s\n", cmdToken[3]);
} else {
    size = lseek(fd_infile, 0, SEEK_END);
    lseek(fd_infile, 0, SEEK_SET);
    pBuffer = malloc(domainbuffer_size);
    if (pBuffer == NULL) {
        fprintf(stderr,"got no memory for download file\n");
        return(CO_E_MEM);
    }
    /* looping through the file */
    commonRet = CO_OK;
    do {
        size = read(fd_infile, pBuffer, domainbuffer_size);
        if (size == 0) {
            break;
        }
        if (size == -1) {
            fprintf(stderr,"error reading download file\n");
            commonRet = CO_SDO_OTHER;
            break;
        }
        /* Download file using default sdo channel */
        fprintf(stderr,"Download %s(%ld bytes) to %x/%d using sdo %d\n",
        cmdToken[3], size,
        index, subIndex, sdo);
        if (executeSendCmd(CMD_SDO_WRITE, index,subIndex,pBuffer,size) != 0)
        {
            commonRet = CO_E_HARDWARE_FAULT;
            break;
        }
    } while(size > 0);
    free(pBuffer);
}
close(fd_infile);
return(commonRet);
```

```
UNSIGNED8 executeSendCmd(
        UNSIGNED16 cmd,      /* command number */
        UNSIGNED16 index,    /* index */
        UNSIGNED8  subIndex, /* sub-index */
        UNSIGNED8  *pBuf,    /* pointer to data */
        UNSIGNED32 size      /* size of data in bytes */
    )
{

    switch(cmd) {
    // .....

        case CMD_SDO_WRITE:
            sdoConError = 0xff;
            if (writeSdoReq(sdo,index,subIndex,pBuf,size) != CO_OK) {
                printout("-- error writeSdoReq(sdo %d, index 0x%x..)\n",
                    sdo, index);
                return 1;
            }
            /* wait for finished sdo transfer
             * the variable sdoConErr can be set in sdoWrCon()
             */
            while (sdoConError == 0xff);

            if (sdoConError == 0)  {
                printout("OK\n");
                return(0);
            } else {
                return 1;
            }
            break;

        // .....
    }
}
```

Listing 12, Domain Example code of an SDO client

4.6.4. SDO Block Transfer

SDO transfers are based on the client-server model with a handshake after each transfer. For a larger block of data this will take a large amount of time. Therefore a new SDO mode has been defined. It is called SDO block transfer.

Using the block transfer a sequence of blocks can be transmitted without a large overhead of handshake each 8 bytes. Each block is a sequence of up to 127 segments (e.g. CAN telegrams) containing only a sequence number and the data.
Each block transfer starts with an initialization phase, where the server and the client can prepare themselves for transferring the blocks and negotiating the number of segments in one block.
There is a finalization phase after transferring the blocks, where the client and server can

optionally verify the correctness of the previous data transfer by comparing checksums derived from the data set.

For the SDO block transfer a Go-Back-n ARQ (Automatic Repeat Request) scheme is used to confirm each block.

If the SDO server does not support SDO block download or upload the client automatically falls back to the traditional segmented transfer.

There are no other programming interface or indication functions to use the CANopen Library with SDO block transfer. Instead block transfers are automatically used for the SDO server and for SDO client if the data to be transferred is greater than or equal to the `#define CONFIG_BLOCK_MIN_DATASIZE` in **cal_conf.h**. It can be set by the *CANopen Design Tool*.

For all transfers the client has to initiate the connection to the server. If the server does not support SDO block transfer the transfer is repeated automatically with segmented transfer. In this case the user is not informed.

During the initialization phase the block size and the usage of CRC checksum are negotiated. Therefore the defines `CONFIG_BLOCK_CRC` and `CONFIG_BLOCK_MAX_CNT` are provided if the CANopen device should support this feature. If `CONFIG_BLOCK_CRC` is set, the client will try to use the CRC generation for transfers. The maximum segment for one block can be set with the define `CONFIG_BLOCK_MAX_CNT`.

If the SDO partner does not support CRC generation or only supports smaller block sizes the values from the SDO partner are used.

All values for SDO block transfer can be set with the *CANopen Design Tool (Light)*.[6]

4.6.5. Dynamic SDO Connections

For configuration tools, analysis tools or HMIs with very intelligent configuration set-up it can be necessary to have several SDO connections to different nodes from time to time in the network. Therefore dynamic SDO connections can be used.
The SDO Manager manages all SDO connections in the network. It can dynamically establish new connections between SRDs (SDO Requesting Device) and a slave (SDO server). Therefore it has an SDO connection table where all established connections are stored.

For each dynamic SDO connection an unused COB-ID from the system is necessary. The SDO Manager also requires information about the COB-IDs that are free in the system. This is configured in the SDO Manager's COB-ID table.

Before an SRD can request dynamic SDO connections it has to be registered at the SDO Manager. This is done with the service "Dynamic SDO Request". If the SDO Manager has received such a request, it scans all nodes for the requested device by reading the object dictionary for all non registered devices. If it has found the requested node a connection between the SDO Manager as server and the SRD as client is established. After that an error control mechanism (Node Guarding or Heartbeat) between the SDO Manager and the SRD is started. If the error control mechanism fails at any time all

[6] *CANopen Design Tool Light* is delivered with the CANopen Library.

established connections from this node are released.

With the established connection, the SRD can request new dynamic SDO connections for other or for all nodes by writing to the object dictionary of the SDO Manager. The SDO Manager establishes the connection by writing the communication parameter in the dictionary of the SRD and the requested slave.

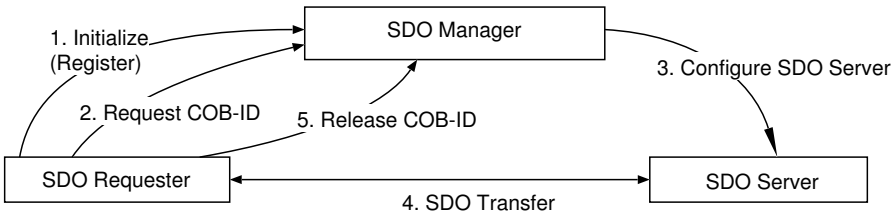


Figure 19, Dynamic SDO Principle

If an SRD no longer needs SDO connections it has to release them. The SRD does this by writing to the SDO Manager's object dictionary. The SDO Manager releases the connection by writing to the object dictionary of SRD and the slave.

The SDO Manager needs the following additional entries in its own object dictionary:

Index	Object	Name	Type	Attr.	Mandatory
$1F00_h$	VAR	Request SDO Connection	UNSIGNED32	wo	yes
$1F01_h$	VAR	Release SDO Connection	UNSIGNED32	wo	yes
$1F02_h$	ARRAY	SDO Manager COB-IDs	UNSIGNED32	rw	yes
$1F03_h$	ARRAY	SDO Connections part 1	UNSIGNED32	ro	yes
$1F04_h$	ARRAY	SDO Connections part 2	UNSIGNED32	ro	no
$1F05_h$	ARRAY	SDO Connections part 3	UNSIGNED32	ro	no
$1F06_h$	ARRAY	SDO Connections part 4	UNSIGNED32	ro	no

Table 20, Objects for an SDO Manager

The following steps are necessary for using the SDO Manager functionality:

- Add the necessary entries for the SDO Manager to your object dictionary.
- Replace and fill out your user reaction for the SDO Manager functionality in the *dynSdoManInd*() in the file **usr_302.c**.
- Fill in all unused COB-IDs from your system into the SDO Manager COB-ID table.
- Initialize the SDO Manager by calling the function *initSdoManager*().

There are two examples in the example directory for the usage of an SDO Manager: **m8**, **m9** for multi-line.

4.6.5.1. SDO Requester

SDO connections are always connections between one server and one client. If more than one node should be connected to another node the connection can not be static. When using dynamic SDOs a node has to be an SDO Requester. Each SRD (SDO

Requesting Device) can request dynamically SDO connections from the SDO Manager.

Before the SRD can request new connections it has to send a registration request (COB-ID 1760) to the SDO Manager. If the request was successful, new connections can be established by writing the communication parameter to the object dictionary of the SDO Manager. Afterwards the SDO Manager sets up the slave and the SRD for the requested connection.

All registered SRDs have to participate in an error control mechanism. If Heartbeat is supported the SDO Manager starts the Heartbeat producer function at the SRD and the Heartbeat consumer function at the SDO Manager. Otherwise from the SDO Manager is used. If the error control mechanism fails at any time, all established connections from the SRD are released by the SDO Manager.

If an SRD not longer needs SDO connections it has to release them. The SRD has to do this by writing to the SDO Manager's object dictionary. The SDO Manager releases those connections by writing to the object dictionary of SRD and the slave.

The SRD needs the following additional entries in its own object dictionary:

Index	Object	Name	Type	Attr.	Mandatory
$1F10_h$	VAR	Dynamic SDO Connection State	`UNSIGNED32`	rw	yes
$1F11_h$	VAR	Slave Failed	`UNSIGNED16`	rw	no

Table 21, Objects for an SDO Requesting Device

The bits at the Dynamic SDO Connection State have the following meaning:

Bit	Name	Description	
0	Request Indication	1 0	dynamic SDO request open no dynamic SDO request or request OK (written by SDO Manager)
1-2	Requested State	1 1 2 2 3	request connection request successful (written by SDO Manager) request all default SDOs request successful (written by SDO Manager) SDO connection to slave established
3	Rec EC		Request Error Control not supported by the CANopen Library
4-7	Reserved		
8-15	Error Code	0 1 2 3 4	No precise details for the reason of the error No SDO channel free for connection from SRD to SDO Manager No more free SDO channels available in the network No more free server SDO entries on slave Slave not available
16-31	OD index		object dictionary index to store the connection

Table 22, Protocol for a Dynamic SDO Connection

The bits of the *Slave Failed* have the following meaning:

Bit	Name	Description
0-7	S Node-ID	Supervised Slave Node-ID
8-15	Reason	The SDO Consumers (SDO Manager) Error Control event

Table 23, Fields of the ***Slave Failed*** object

The following steps are necessary in order to use the functionality:

- Add the necessary entries for SDO Requester to your object dictionary.
- Initialize the SDO request variables by calling the function *initDynSdoAccess*().
- Start the registration by calling the function *writeDynSdoRegister*().
- Test the state of the registration (see object $1F10_h$).
- Request dynamic connections by calling the function *writeDynSdoRequest*().
- Release dynamic connections by calling the function *writeDynSdoRelease*().

There are two examples in the example directory:

s8 simple program to register and request a dynamic SDO to node 32

s9 interactive program to request and release dynamic SDO connections and read and write values at the connected slave

4.7. PDO Usage

PDOs serve for transferring real time data without overhead. Contents of the data which should be transmitted must be determined before - this is called mapping. The mapping can be set at compile time (static mapping) or at run time (dynamic mapping).
With dynamic mapping, memory is reserved for the maximum mapping by the initialization.

The mapping can take place bit-wise or byte-wise. The bit-wise mapping is necessary for variables unequal to 8, 16 or 32 bit (i.e. bit variables) and without holes at the CAN transmission. If variables are to be mapped bit-wise, the define `CONFIG_BIT_ENCODING` must be set. This requires, however, larger code blocks and a longer processing time for PDO than the byte-wise mapping.

Before usage all PDOs must be defined. A maximum of 512 Receive PDOs and 512 Transmit PDOs with a maximum mapping of 64 entries for each direction are possible. Initialization is done with the function *definePdo*() (Listing 13).

```
definePdo(5, CONSUMER, CO_FALSE);       // define PDO 5 as consumer

/* setup COB-ID is necessary for PDOs 5..512 */
setCobId(0x1404, 1, 0x67f);             // set COB-ID
```

Listing 13, Example for *definePdo*()

During the initialization of the CANopen Library a *Reset Communication* will be executed and all entries of the object dictionary are set to their default values. For the first 4 Receive PDOs and the first 4 Transmit PDOs the default values for the COB-IDs are calculated according to the Predefined Connection Set and the actual node-ID. The new COB-IDs are entered into the object dictionary, independently of the default values of the object dictionary generated by the *CANopen Design Tool*.

For fast access at run time all PDO data and the addresses of the mapping variables are stored in internal administrative structures. For all modifications of the communication parameters in the object dictionary the function *setCommPar*() has to be called in order to update the internal structures. Changes from a remote node via SDO automatically update the internal structures.

To change the PDO mapping data first the PDO has to be disabled by setting bit 31 of the COB-ID. Then the mapping has to be deactivate by writing 0 to the sub-index 0 of the mapping data. This sub-index 0 always determines the valid numbers of objects that have been mapped or not, as with all other indexes the number of sub-indexes. After entering the new mapping data, the number of valid mapping entries has to be updated, and the function *setCommPar*() has to be called in order to update the internal structures. Finally the PDO has to be enabled again by clearing bit 31 of the COB-ID.

```
/* disable pdo */
setCobId(0x1400, 1, PDO_NO_VALID_BIT);    // set cob-id to no valid

/* deactivate mapping */
mapCnt = 0;
putObj(0x1600, 0, &mapCnt, 1, CO_TRUE);   // set value to object dict.
setCommPar(0x1600, 0);                    // set internal values

/* set new mapping */
mapEntry = 0x20000120;                    // map 2000:1 U32
putObj(0x1600, 1, &mapEntry, 4, CO_TRUE); // set value to object dict.
mapEntry = 0x21000108;                    // map 2100:1 U8
putObj(0x1600, 2, &mapEntry, 4, CO_TRUE); // set value to object dict.

/* validate mapping */
mapCnt = 2;                               // two mapping entries
putObj(0x1600, 0, &mapCnt, 1, CO_TRUE);   // set value to object dict.
setCommPar(0x1600, 0);                    // set internal values

/* enable pdo */
setCobId(0x1400, 1, 0x220);               // set cob-id
```

Listing 14, Example for Changing Mapping Parameter

Transmitting PDOs is done using the function *writePdoReq*(). Only the PDO number is given as function argument. The CANopen Library composes automatically the transmit buffer by saving the mapped data at the transmit buffer. Asynchronous PDOs are transmitted immediately, synchronous PDOs are stored and transmitted after the next received SYNC message.

```
writePdoReq(1);                 // write PDO 1
writePdoReq(3);                 // write PDO 3
```

Listing 15, PDO Transmission

PDOs are transmitted with high priority. To avoid blocking of the CAN communication by high priority PDOs an inhibit time parameter can be defined. The inhibit time is a minimum time between the 2 consecutive transmissions of a PDO. If the inhibit time has not elapsed yet, the function *writePdoReq*() returns an error code. The application can try to send it later or is waiting until the inhibit time is elapsed (Listing 16).

```
do
{
    ret = writePdoReq(1);
    FlushMBox();                // do other CANopen tasks
} while (ret == CO_E_INHIBITED)
```

Listing 16, PDO Inhibit Time Waiting

Of course, this loop should not block the whole application. Please consider that the inhibit time can be set via network from a configuration tool and is therefore not determined by the application. It can last more than 6 sec as maximum.

If the SYNC period is too small or there is no SYNC in the network, PDO can also be sent time driven by specifying an event timer. The event timer can be used for asynchronous PDO only. If the entry for the event timer is greater than zero the PDO is transmitted cyclically with this rate. Before the PDO is transmitted, the indication function *pdoEventTimerInd*() is called. The application can update the value in the object dictionary before the data are filled into the transmit buffer.

For Receive PDOs the event timer can also be used. If the event timer is unequal zero it is restarted every time a PDO was received. If the timer is up the indication *pdoTimerInd()* is called. The application can now request the PDO.

PDOs can be requested by other nodes via RTR[7]. This request is handled by the CANopen Library. Before the transmit data are generated, the indication function *rtrPdoInd()* can be used to update the data at the object dictionary.

A positive acknowledgment of the function *writePdoReq*() does not mean that the telegrams are sent successfully. It means only that a successful entry into the transmit buffer was done. In the case of errors (node not in state OPERATIONAL, inhibit time has not expired) the function returns with the appropriate error code and does not transmit the data.

The reception of PDOs is indicated with the function *pdoInd*(). All mapped data for this PDO are written to the object dictionary before this function is called. If the number of received data for the defined mapping is not sufficient, the entire PDO is rejected and an *Emergency Message* is generated. In this case no data are stored at the object dictionary.

[7] Please refer to the document CiA-AN802 - "CANopen statement on the use of RTR-messages" for use of RTR messages.

```
/************************************************************
*
* pdoInd - PDO indication function
*
* sends PDO 2 on line 1 if PDO 1 has been received on CAN line 0
*
* \returns
* nothing
*/
void pdoInd(
     UNSIGNED16 pdoNr  /* number of PDO */
#ifdef CONFIG_MULT_LINES
     ,UNSIGNED8 canLine     /* number of CAN line
                               0..CONFIG_MULT_LINES-1 */
#endif
     )
{
    /* static mapped PDO */
    if (pdoNr == 1) {
        /* data already stored at OD */
        user_action();
    }

    /* dynamic mapped  PDO */
    if (pdoNr == 2) {
        /* check, if variable setpoint is mapped */
        if (getMapObjAddr(pdoNr, 1) == &setpoint)  {
            /* yes, is mapped */
            user_action2();
        }
    }
}
```

Listing 17, PDO Indication

If dynamic PDO mapping is enabled, a detection of the contents of the PDO can be made in the function *pdoInd*(). One variant is shown in figure 20.

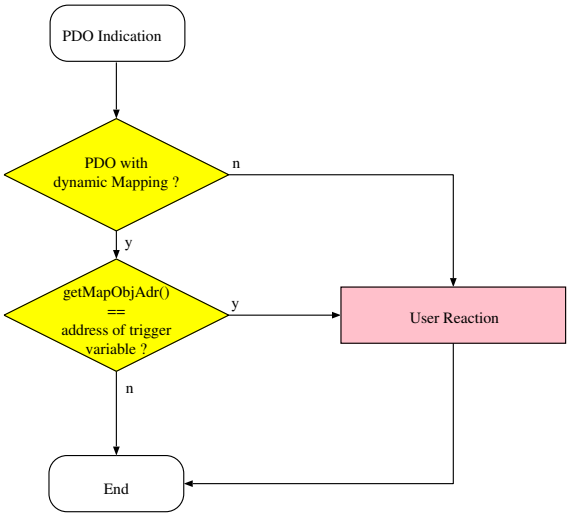


Figure 20, The PDO Indication Flow

4.7.1. Multiplexed PDO Usage

If the application has a lot of data with the same properties a special PDO type can be used. It is called a *Multiplexed PDO* (MPDO). MPDOs transmit with every CAN message the index and the sub-index of the given data. Therefore the maximum data length can be only 4 bytes. The transmitted index and sub-index can be the index and the sub-index of the producers object dictionary (MPDO Source Addressing Mode) or the index and the sub-index of the consumers object dictionary (MPDO Destination Addressing Mode).

The CANopen Library uses the same functions for both modes. The initialization is done with the default PDO initialization function *definePdo*(). For writing MPDOs the function *writeMPdoReq*() has to be used. For the MPDO destination addressing mode the parameter *node* is not necessary and should be 0.

If an MPDO is received the indication function *mpdoInd*() is called. It works just as the *pdoInd*() function.

The usage of MPDOs allows the transmission of several homogeneous PDOs with a minimum of mapping and communication parameter entries in the object dictionary. Since the mapping is not constant, a longer processing time is necessary for creating or analyzing the CAN messages.

4.7.1.1. Destination Address Mode

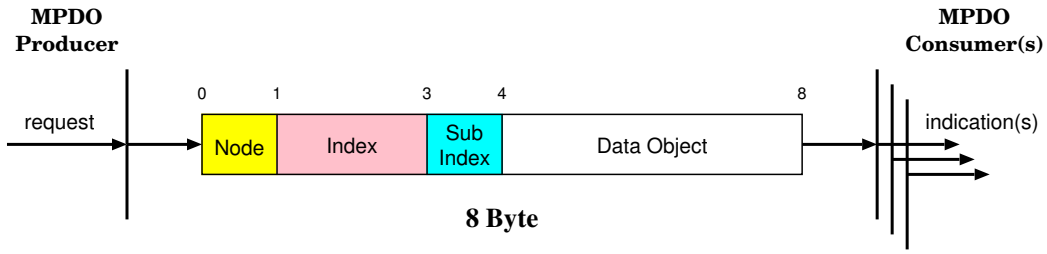


Figure 21, Structure of an MPDO in Destination Address Mode

In destination address mode index and sub-index refer to the consumer. This allows access to the consumers' object dictionary in an SDO-like manner. When the destination node is 0 it allows a broadcasting to write into the object dictionary of more than one node simultaneously without sending a PDO for each single node.

4.7.1.1.1.

Entries in the object dictionary:

Index	Sub-Index	Description	Value
$18xx_h$		communication parameter	
$1Axx_h$	0	number of mapping entries	255
$1Axx_h$	1	mapping entry	application specific

Table 24, Objects for an in Destination Address Mode

Function:
writeMPdoReq(pdoNumber, dest.-node, dest.-index, dest.-subIndex)

4.7.1.1.2.

Entries in the object dictionary:

Index	Sub-Index	Description	Value
$14xx_h$		communication parameter	
$16xx_h$	0	number of mapping entries	255

Table 25, Objects for an in Destination Address Mode

If an MPDO was received, the data will have been written into the received index and sub-index.

4.7.1.2. Source Address Mode

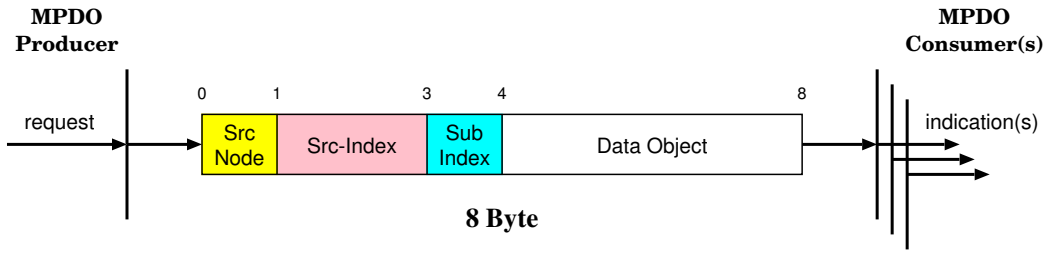


Figure 22, Structure of an MPDO in Source Address Mode

In source address mode index and sub-index refer to the producer. The transmission type has to be either 254 or 255.

4.7.1.2.1.

Entries in the object dictionary:

Index	Sub-Index	Description	Value
$18xx_h$		communication parameter	
$18xx_h$	2	transmission type	254 or 255
$1Axx_h$	0	number of mapping entries	254
$1FA0_h - 1FCF_h$	0-254	object scanner list	

Table 26, Objects for an in Source Address Mode

The producer uses an object scanner list to configure which objects have to be sent.

Each scanner list entry has the following format:

MSB		LSB
31-24	23-8	7-0
Block Size	Index	Sub-Index

Table 27, Entry in Scanner List

Each table entry describes an object that can be sent via MPDO. It is possible to describe consecutive sub-indexes by setting the parameter block size to the number of sub-indexes.

Function:
writeMPdoReq(pdoNumber, 0, src-index, src-subIndex)

Only one producer MPDO of this type is allowed for each node.

4.7.1.2.2.

Entries in the object dictionary:

Index	Sub-Index	Description	Value
$16xx_h$	0	number of mapping entries	254
$1FD0_h$–$1FFF_h$	0–254	dispatch entry	

Table 28, Objects for an in Source Address Mode

The consumer uses an object dispatcher list as a 'cross reference' between the remote object of the producer and the local object dictionary.

Each dispatch entry has the following format:

MSB					LSB
63-56	55-40	39-32	31-16	15-8	7-0
Block Size	Local Index	Local Sub-Index	Prod Index	Prod Sub-Index	Prod Node

Table 29, Entry Dispatcher List

If a PDO was received, and the node-ID of the producer, index and sub-index match an entry in the dispatcher list, then the data is written into the local object dictionary in the index and sub-index given in this entry.
The parameter "block size" allows the description of consecutive sub-indexes to be used.

For example: if sub-index 1-9 of the PDO producer should be mapped to sub-index 11-19 of the local node, this range is defined by:

Producer sub-index = 1
Local sub-index = 11
Block Size = 9

Non configured entries shall have the value 0.

4.7.1.2.3. Application Notes for

If this object is written with invalid data by an SDO transfer then an abort domain transfer is generated. Because the data size is greater than 4 bytes it can not be saved and the invalid data will remain in the object dictionary.

The generation of the object dictionary can be created by the *CANopen Design Tool*.

If dynamic mapping is used and a PDO is defined as a multiplexed PDO by the function *definePdo*() it can not be configured as a normal PDO.

4.8. Emergency

Emergency messages serve for transmitting and receiving error messages. The initialization can take place for one producer and up to 127 consumer. The function *defineEmcy*() with the appropriate parameter initializes the emergency service for producer or consumer. If the emergency consumer list in the object dictionary at index 1028_h exists, then all entries with a valid COB-ID are initialized automatically. If it does not exist the emergency consumers can be added by the function *setEmcyConsumerCobId()*.

```
// producer
retVal = defineEmcy(PRODUCER);

// consumer
retVal = defineEmcy(CONSUMER);

// if emcy consumer list does not exist
// add node 5
retVal = setEmcyConsumerCobId(5, 0x185);

// add node 35
retVal = setEmcyConsumerCobId(35, 0x1a3);
```

Listing 18, Example for *defineEmcy*()

Emergency messages on the bus are generated by the function *writeEmcyReq*() (Listing 19). The CANopen Library generates automatically an entry in the predefined error field (Index 1003_h), sets the general error bit at the object 1001_h, creates the emergency telegram and forces the transmission. Object 1003_h, the *pre-defined error field* is an array of UNSIGNED32. *writeEmcyReq*() stores the error code in the two lowest bytes and two bytes of the manufacturer specific error code in the two upper bytes. If more of the error bits are to be set in the index 1001_h then the application has to do this before *writeEmcyReq*() is called.

```
UNSIGNED8 manuErr[5];
RET_T ret;

manuErr[0] = 0x1;
manuErr[1] = 0x2;
manuErr[2] = 0x3;
manuErr[3] = 0x4;
manuErr[4] = 0x5;

ret = writeEmcyReq(0xffff, &manuErr[0]);
if (ret != CO_OK)  {
    printf( "error EMCY 0xFF00 %d",(int)ret);
}
```

Listing 19, Example for *writeEmcyReq*()

Each error is saved in the error list at index 1003_h sub-index 1. All other already available errors are shifted automatically from one sub-index to the next sub-index. The sub-index 0 of the error list (index 1003_h) always indicates the number of errors. If the error list is full and a new error occurs then the oldest error in the error list is cleared automatically.

Writing in the sub-index 0 in the error list is permitted only with the value 0. Thus all entries in the error list are deleted with the help of the function *eraseErr*(). Resetting the general error bit and the other error-specific bits in the error register index 1001_h must take place in the application.

For transmitting Emergency messages an inhibit time can be specified. After each modification of this time the function *setCommPar*() has to be called in order to update the

internal structures.

The reception of error messages from other nodes in the network is indicated by the function *emcyInd*(). This function makes all data contained in the emergency message available to the application. Storage of the data does not take place.

```
/***************************************************************************
*
* emcyInd - indicates the occurrence of an emergency object
*
* In this function the user has to define his application specific
* error handling. The function must send a message to the server in
* order to repair the error.
*
* \returns
* nothing
*
*/
void emcyInd(
        UNSIGNED8       emcyNode,       /* emergency node */
        EMERGENCY_T     *pEmcy          /* emergency message */
#ifdef CONFIG_MULT_LINES
        ,UNSIGNED8 canLine              /* number of CAN line
                                           0..CONFIG_MULT_LINES-1 */
#endif
    )
{
    switch (pEmcy->errCode & 0xFF00) {
        case 0x4000:
            printf("Temperature\n");
            break;
        case 0x5000:
            printf("Device Hardware\n");
            break;
        case 0x6000:
            printf("Device Software\n");
            break;
        case 0xFF00:
            printf("User specific\n");
            break;
        default:
            printf("emergency ind %x\n",pEmcy->errCode);
            break;
    }
    printf("errReg: %x, Manu: %x %x %x %x\n",
        pEmcy->errReg, pEmcy->manu[0], pEmcy->manu[1], pEmcy->manu[2],
        pEmcy->manu[3], pEmcy->manu[4]);
}
```

Listing 20, Emergency Indication

The indication function is called only for the error messages, which have a valid COB-ID entry at the object dictionary list or was added by *setEmcyConsumerCobId()* before.

4.9. SYNC Usage

The SYNC telegram serves for synchronous transfer of PDOs and synchronous execution of internal procedures in different nodes of the network. A node can either be the SYNC producer or the SYNC consumer. The type of service must be determined by the initialization of the function *defineSync*() or by setting the appropriate bit at index 1005_h. Additionally, the SYNC cycle time has to be set for the SYNC producer in the object dictionary and the internal structures have to be updated with the function *setCommPar*().

```
// define sync producer
retVal = defineSync(PRODUCER);

// set new cycle time
cycleTime = 1000;                          // set new cycle period
putObj(0x1006, 0, &cycleTime, 4, CO_TRUE); // set value to object dict.
setCommPar(0x1006, 0);                     // update internal values

// start sync
startSyncReq();
```

Listing 21, Define SYNC producer

The SYNC telegram is transmitted automatically according to the given time, if the producer bit is set. With the arrival or the transmission of the SYNC telegram the synchronous PDOs are assembled and transmitted. Data that were received with the last SYNC are copied to the object dictionary. For each received PDO also the PDO indication function *pdoInd*() is called similar to the PDOs received asynchronously.

For the synchronization of the different nodes in the network two user functions are available: *syncPreCommand()* and *syncCommand()*. The first function is called immediately after the SYNC message was received or transmitted, and the second is called after all functionality for the SYNC process was done.

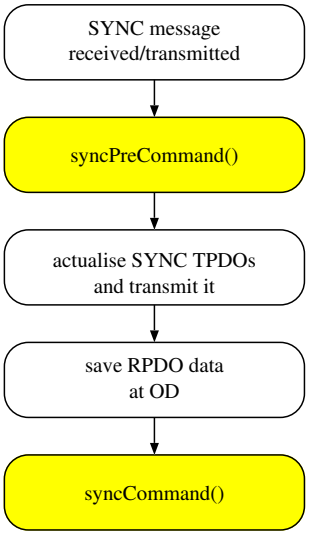


Figure 23, SYNC Process

Sending of SYNC messages can be started by setting the appropriate bit for the SYNC producer COB-ID at the object dictionary or by the function *startSyncReq()* or *stopSyncReq()*.

4.10. Error-Control-Mechanisms

CANopen defines two error control mechanisms. These mechanisms are called Node Guarding and Heartbeat. Each node has to provide at least one service. Even if both services are implemented guarding has to be done with only one service. Sending of guarding messages and guarding of other nodes with the Heartbeat service is possible for master and slave devices. Guarding with the Node Guarding service is possible as NMT master device. Only services that have been initialized can be used.

```
// init node only for heartbeat usage
createNodeReq(CO_FALSE, CO_TRUE);

// init node only for guarding usage
createNodeReq(CO_TRUE, CO_FALSE);

// init node for guarding and heartbeat usage
createNodeReq(CO_TRUE, CO_TRUE);
```

Listing 22, Create Node Request

Boot-up messages of all devices can be received from a Heartbeat consumer and from a Node Guarding master. It is not necessary to setup guarding service for any special device.

All events for Heartbeat and Node Guarding master monitoring are signaled by the indication functions *mGuardErrInd*() (Listing 23) and the Node Guarding slave are signaled by *sGuardErrInd*() (Listing 24).

4.10.1. Node Guarding

The Node Guarding protocol is based on a master slave relationship. The master requests the current state of the slave cyclically with an RTR telegram. The slave answers this telegram with its state and an additional toggle bit. The Node Guarding is started on the master with the function *startNodeGuardReq*() and operates independently in the background. If the RTR telegram is not answered by the slave within the inquiry cycle, the function *mGuardErrInd*() (Listing 23) with the parameter **CO_LOST_GUARDING_MSG** is called. If the slave does not transmit responses according to adjusted lifetime factor, the guarding becomes inactive and the user is informed by the function *mGuardErrInd*() (Listing 23) and the parameter **CO_LOST_CONNECTION** (Listing 24).

```
/***************************************************************************
*
* mGuardErrorInd - indicates the occurrence of a node-guarding error
*
* This function defines the reaction for node-guarding error
* which is indicated at the master.
* This function must be short, because it will be called from a
* interrupt service routine in plain systems.
*
* \returns
* nothing
*
*/

void mGuardErrorInd(
     UNSIGNED8 nodeId,  /* Node-Id of error source */
     ERROR_SPEC_T kind  /* kind of error */
#ifdef CONFIG_MULT_LINES
     ,UNSIGNED8 canLine     /* number of CAN line
                               0..CONFIG_MULT_LINES-1 */
#endif
   )
{
  switch(kind) {
    case CO_LOST_GUARDING_MSG:
        printf("LOST_GUARDING_MSG node %u\n",(unsigned int)nodeId);
        break;
    case CO_LOST_CONNECTION:
        printf("LOST_CONNECTION node %u\n",(unsigned int)nodeId);
        break;
    case CO_LOST_HEARTBEAT:
        printf("LOST_HEARTBEAT node %u\n",(unsigned int)nodeId);
        break;
    case CO_HB_STARTED:
        printf("HB_STARTED node %u\n",(unsigned int)nodeId);
        break;
    case CO_BOOT_UP:
        printf("BOOT_UP node %u\n",(unsigned int)nodeId);
        break;
  }
}
```

Listing 23, Master Guarding Indication

The Node Guarding on the master can be started only if the guarding time and lifetime are larger than 0.

The guarding services are independent from NMT master services. Normally, the NMT master takes over the task of Node Guarding the remote nodes. That way the NMT master and the Node Guarding master are setup at the same time. For this purpose the function *addRemoteNodeReq()* is available. It can also be used to initialize the nodes that should be guarded. Otherwise nodes can be added to the guarding service with the function *addGuardingSlave()*. In order to change guarding parameters the function *setGuardTimePara()* is available.

Likewise the slave can monitor the queries of the masters (life guarding). The life guarding starts with the reception of the first RTR request by the master. If the RTR queries are missing, the function *sGuardErrInd*() is called.

```
/*****************************************************************
*
* sGuardErrorInd - indicates the occurrence of a node-guarding error
*
* This function defines the reaction for node-guarding error
* which will be indicated at the local slave.
* This function must be short, because it is called from an
* interrupt service routine in plain systems.
*
* \retval 0 node should remain in the current state
* \retval 1 node should be forced to the state PRE_OPERATIONAL
*
*/

UNSIGNED8 sGuardErrorInd(
          ERROR_SPEC_T kind  /* kind of error */
#ifdef CONFIG_MULT_LINES
         ,UNSIGNED8 canLine     /* number of CAN line
                                   0..CONFIG_MULT_LINES-1 */
#endif
          )
{
    switch(kind) {
        case CO_LOST_GUARDING_MSG:
            return 0;
        case CO_LOST_CONNECTION:
            return 1;
        default:
            return 0;
    }
}
```

Listing 24, Slave Guarding Indication

The application can determine which communication state the node should have after calling this function. If the return value equals one, the communication state is changed to PRE-OPERATIONAL. Otherwise it is left in its current state.

4.10.2. Heartbeat

The Heartbeat service allows each node to monitor every other node in the network. Each Heartbeat producer transmits cyclically its own Heartbeat (Heartbeat producer). The monitoring can now take place from one or more nodes (Heartbeat consumer). Each node can be simultaneously producer and consumer.

The Heartbeat producer starts the cyclic transmission of its own Heartbeat message immediately if the entry in the object directory 1017_h is greater than 0. The Heartbeat consumer starts the monitoring automatically after the reception of the first Heartbeat

message. Through the function *mGuardErrInd*() (Listing 23) the application is informed about each new state. This indication function is called with the arrival of the boot-up message, the start of Heartbeat messages, and if Heartbeat messages are missing.

The initialization of the Heartbeat consumer is carried out with the function *defineHeartbeatConsumer*(). All nodes of the Heartbeat consumer list (Index 1016_h) are initialized for Heartbeat guarding. Changes to Heartbeat parameters can be performed directly in the object dictionary. After each change the internal variables have to be updated with the function *setCommPar*().

```
UNSIGNED32 monPara;

// set new monitoring time 0x123 for node 0x44
monPara = (UNSIGNED32) (0x44 << 16) | 0x0123;
putObj(0x1016, 3, &cycleTime, 4, CO_TRUE); // set value at sub-index 3

setCommPar(0x1016, 3);                     // update internal values



// change monitoring time to 0x222
monPara = (UNSIGNED32) (0x44 << 16) | 0x0222;
putObj(0x1016, 3, &cycleTime, 4, CO_TRUE); // set value at sub-index 3

setCommPar(0x1016, 3);                     // update internal values
```

Listing 25, Set Heartbeat Parameter

4.11. NMT Service Usage

Network Management Services (NMT) serve for switching communication states of CANopen nodes. Only the CANopen master is allowed to send NMT commands in the network.
The NMT master administers the communication states of all nodes in the network. Therefore the master must create appropriate administrative structures with the function *createNetworkReq*() (Listing 26). Each node in the network where the NMT master will send NMT commands to has to be registered using the function *addRemoteNodeReq*().

```
// create a network structure for using Heartbeat
createNetworkReq();

// add remote node 12, Heartbeat Time of 500 msec
addRemoteNodeReq(12, 500, 0, CO_TRUE, CO_FALSE);

// add remote node 42, Heartbeat Time of 800 msec
addRemoteNodeReq(42, 800, 0, CO_TRUE, CO_FALSE);
```

Listing 26, Create Network Request

The guarding functions Node Guarding and Heartbeat can be used independently from NMT services with the NMT functions.

NMT commands can be transmitted either for individual nodes or for the entire network. The functions *startRemoteNodeReq*(), *enterPreopState*() and *stopRemoteNodeReq*() are available for this task. Node-ID '0' is used to address all CANopen nodes.

```
// start all nodes in the network
startRemoteNodeReq(0);

// stop the node 7
stopRemoteNodeReq(7);

// enter PER-OP node 8
enterPreOpStateReq(8);
```

Listing 27, Usage of NMT Services

Furthermore there is the function *newStateInd*(). This function informs the user application about each transition of the communication state machine. This information can be important, because a few communication services are not available in certain states. For example, the application transmits error codes via PDO and the master forces the node to PRE-OPERATIONAL. Then PDOs are not allowed.

If the device can not change to the state OPERATIONAL it can signal this by the return value. In this case the node stays in the current state.

The reset behavior is a property only for slaves. Each slave can receive a *Reset Application* or a *Reset Communication* command from the master. The module **nmtslave.c** contains function templates for *resetApplInd*() and *resetCommInd*(). Within these functions an individual reset of the application data and states can be carried out. In the function for the communication reset the device can change the CAN bit rate (only for single device networks) or the node-ID.

4.12. Flying Master Usage

4.12.1. Common Hints

The flying master principle was original specified for maritime applications in the standard CiA-307. Now some more services are defined, they can be performed only from one node in the network. In most cases the CANopen master has to realize this functionality. A loss of the CANopen master can be fatal for the whole network.
For this reason the flying master principle was moved to the CiA-302 as a common service.

The flying master principle is based on Heartbeat monitoring of all master capable devices in the network. If the active master fails a new negotiation is started automatically, depending on a priority of nodes and the node number of the master capable devices. The master negotiation is won by the node with the highest priority (prior = 0) and the lowest node-ID.) The active master monitors the network cyclically for other active masters and starts a new master negotiation if necessary.

4.12.2. Flying CANopen Master Functionality

All Flying Master devices have to implement index $1F80_h$ where bit 0 and 5 is set. Otherwise they can not participate in the master negotiation process.

4.12.2.1. CANopen Master Boot-up Process

- After boot-up the detection of an active master starts.
- If no master is available
  - at power on reset a *Reset Communication* for all devices is transmitted
  - start new master negotiation process

  If the priority of the active master is less than its own priority a *Force Reset Communication* will be sent to the active master. The active master will then send the *Reset Communication* command to all devices.
- If the calculated waiting time is done and no other master id was received the master sends its own master-ID and works as network master otherwise it works as network slave.

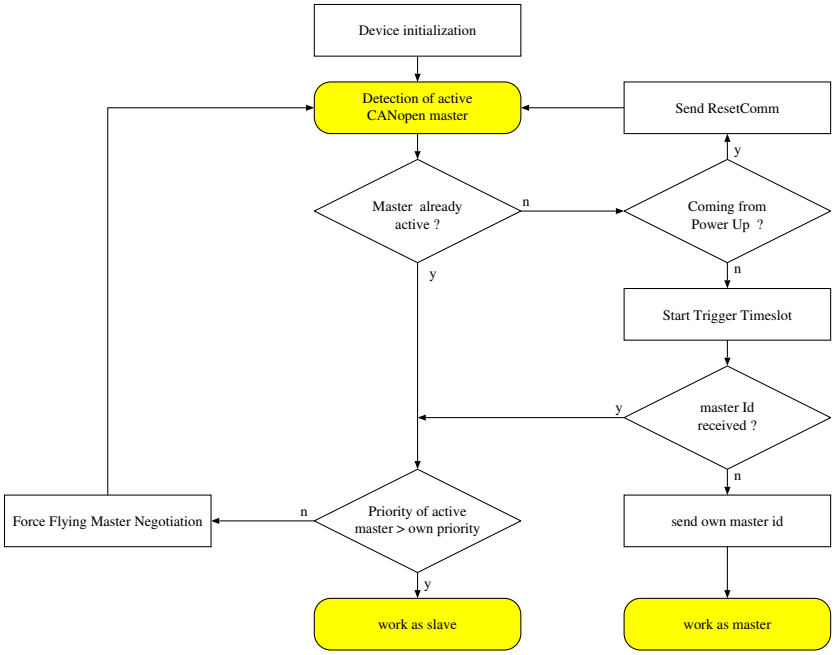


Figure 24, Flying Master Boot Process

4.12.2.2. Detection of an active CANopen Master

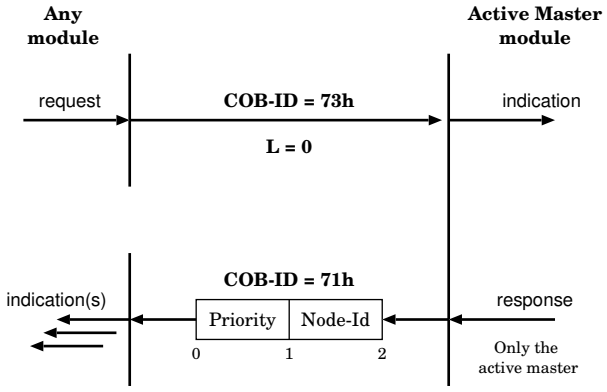


Figure 25 Detection of an Active Master

- The protocol is initiated by any master capable node in the network.
- Only the active master responds with its node-ID and priority.
- If the master does not answer within the timeout period (index $1F90_h$, sub-index 1), a new master negotiation process is started. The default timeout value is 100 ms.

4.12.2.3. Master Negotiation

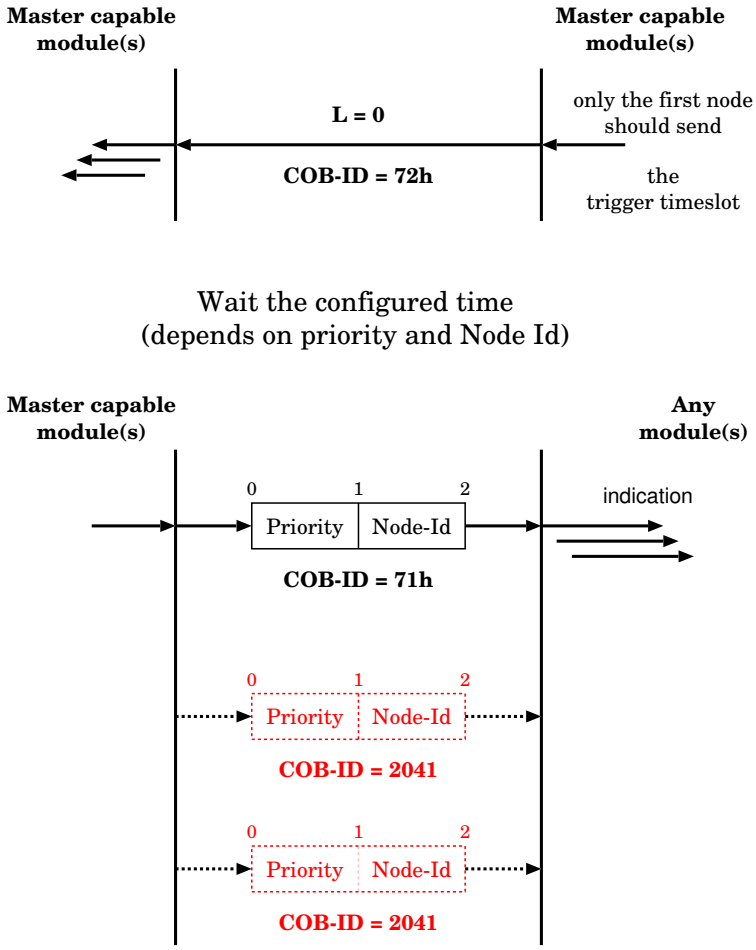


Figure 26 , Master Negotiation

- The master negotiation starts by sending a *Reset Communication*, triggered by the active master.
- After the *"Detection of an active master"* time a master capable device sends an *ID Trigger Timeslot*.
- This trigger will start a local timer at all master capable modules.
- After the timer is done the master-ID is sent if no other master-ID was received before.
- The time is calculated by priority and the node number.

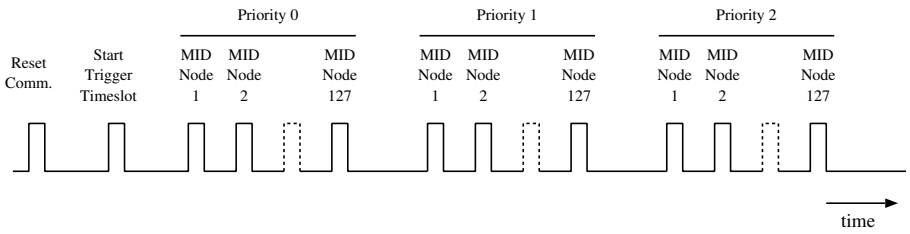


Figure 27 , Master Negotiation

- The active CANopen master cyclically asks for other active masters. When the master receives a reply a new master negotiation is forced.

4.12.2.4. Force Master Negotiation

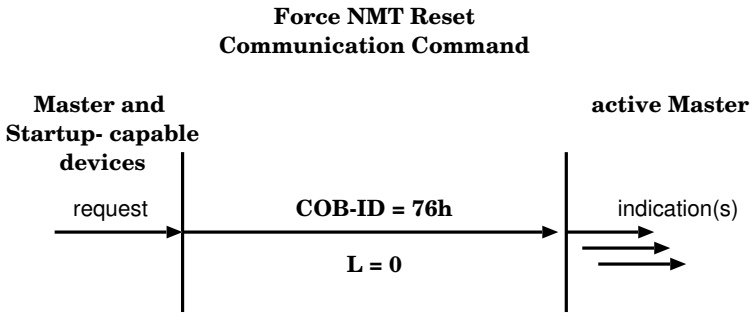


Figure 28 , Force Master Negotiation

- The protocol is started by a master capable device.
- Master negotiation starts again.

4.12.2.5. Detecting CANopen Master Capable Devices

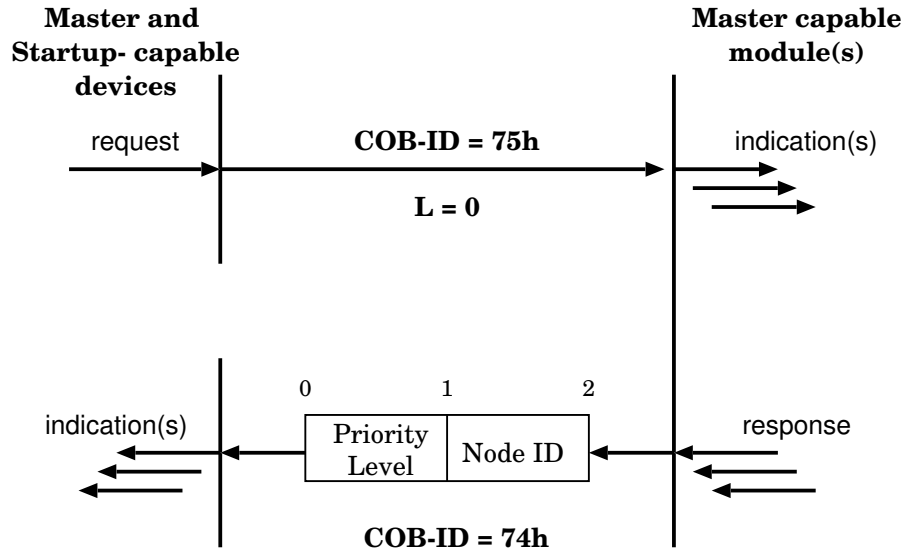


Figure 29 , Detecting CANopen Master Capable Devices

- The protocol is initiated by any node in the network by sending the MCREQ (Master Capable Request) protocol.
- Each Master Capable Device answers with a MRESP (Master Response) protocol. Because the protocol message does not contain any data all nodes can send it simultaneously.

4.12.3. Application Programming Interface

Function	Description
defineFlyingMaster()	Flying Master initialization
startFlyingMaster()	start Flying Master usage
detectMasterReq()	Detecting Master Capable Devices
activeMasterReq()	Detection of a active system master
forceCommResetReq()	force Communication Reset from active master
flyingMasterInd()	indication function for Flying Master events

Table 30 , API for the Flying Master Functionality

A new master negotiation is started by the CANopen Library when:

- Heartbeat is failing for the active master
- an unknown or a second master is detected
- for master with lower priority

4.13. Redundant Communication

The redundant communication is designed to fulfill the requirements of high reliable systems (e.g. maritime or medical applications). It is based on a communication using two separate physical CAN wires. The first line is called the *Default-CANline* and the second one *Redundant-CANline*. The communication starts on the *Default-CANline*. If the line is disturbed or fails the communication switches to the *Redundant-CANline*. If the *Default-CANline* has recovered from failure the communication will switch back to this line. A prerequisite of it is the active Heartbeat monitoring of all nodes on both lines in the network.

The usage of non-redundant nodes is possible too. This kind of nodes have to be connected only to one line.

The redundant communication is described in CiA-302.

4.13.1. Line Switching

Line switching can be done automatically by the CANopen Library or by the application. Before the CANopen Library performs a line switching it calls the user indication function *redundancyInd()*. Within this function the user application can avert the automatic line switch by returning with a special return code.

4.13.1.1. Line Negotiation at Boot-up

After a power on reset and after a *Reset Communication* a line negotiation is performed by the following steps:

- A timer will be started.
- If 3 Heartbeat messages are received from all redundant nodes on the default line then the *Default-CANline* will become the active line.
- The timer is stopped, if an *active line* message was received.
- If the timer expires then the *Redundant-CANline* will become the active line.

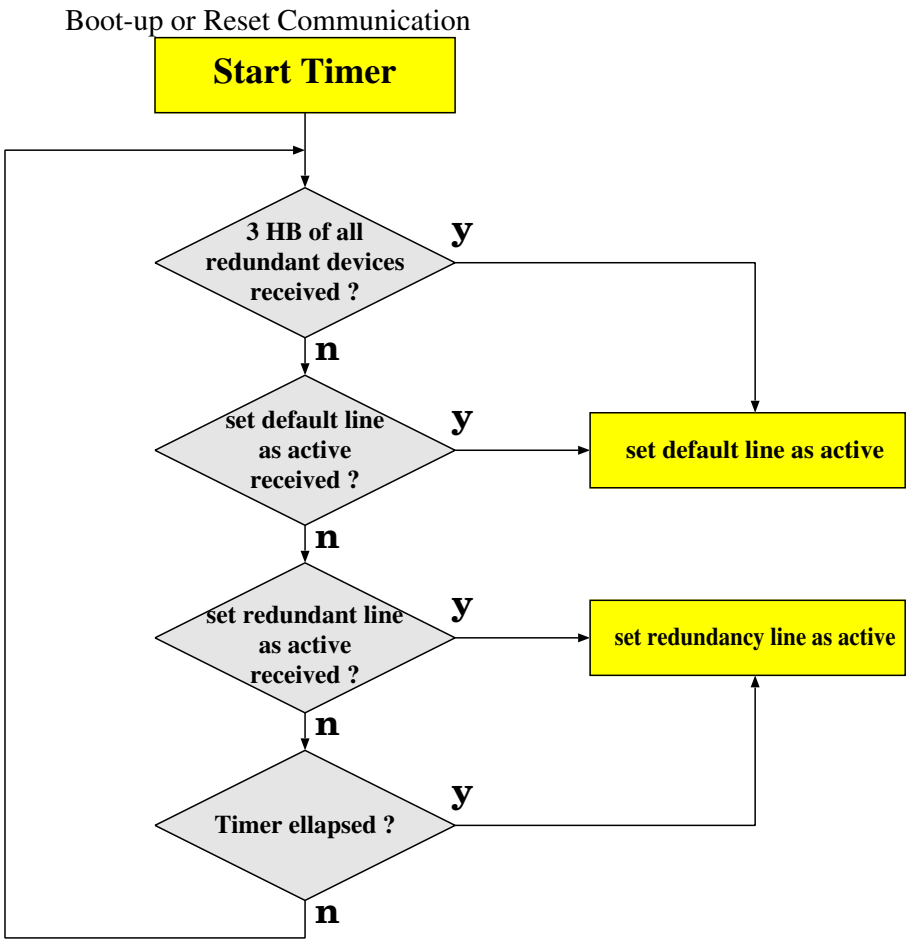


Figure 30, Program Flow after Boot-up

4.13.1.2. Line Monitoring

If the *Default-CANline* is active all other nodes will be monitored by Heartbeat. If the Heartbeat from one of the other nodes fails the user indication function *redundancyInd()* is called. If it returns with `CO_TRUE` then a switch to the *Redundant-CANline* will be performed. If the command *active line* is received on the *Redundant-CANline* a switch to this line is performed without calling the user indication.

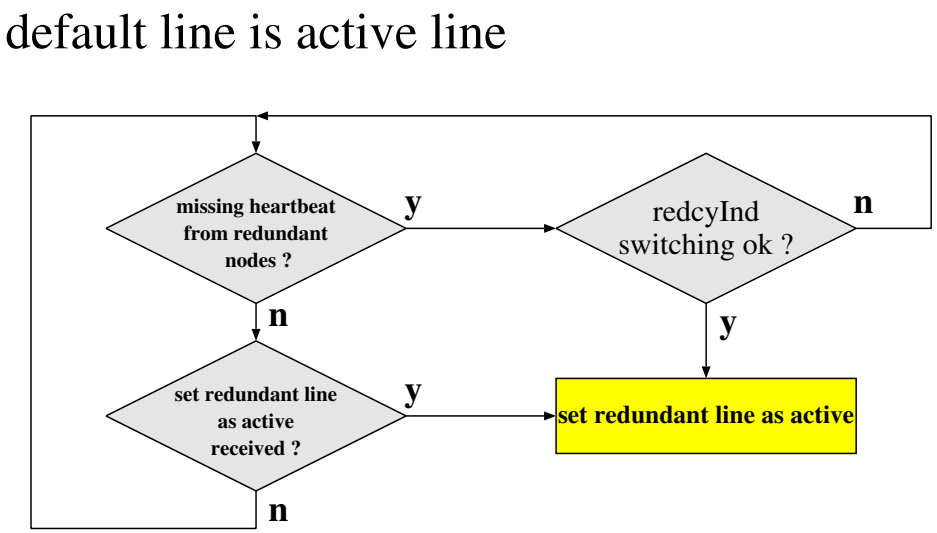


Figure 31, Program Flow when Default CAN Line has Errors

If the *Redundant-CANline* is active the CANopen Library checks for 3 error-free received Heartbeat from all redundant nodes on the *Default-CANline*. If this is the case, the user indication *redundancyInd()* is called and a switch back to the *Default-CANline* will be performed if the return value is `CO_TRUE`.

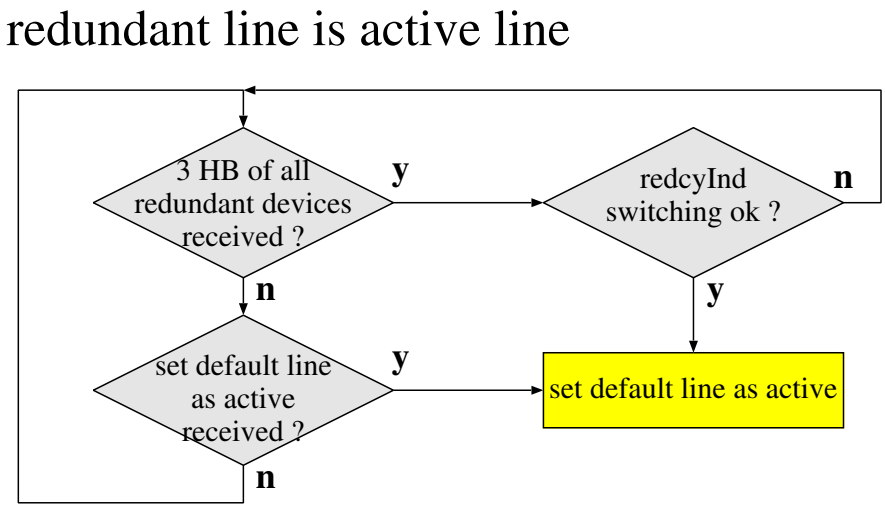


Figure 32, Program Flow when Default CAN Line is Error-Free

4.13.2. Message Transmission

Message transmission depends on the used service, the active CAN line and the actual communication state of this line:

Service	Transmit On	Condition	Treatment
NMT	any	nothing	line depending
NMTErr	any	depends on line state	line depending
PDO	both	OPERATIONAL	both lines
EMCY	both	OPERATIONAL or PREOP	both lines
TIME	both	OPERATIONAL or PREOP	active line
SYNC	both	OPERATIONAL or PREOP	active line
Server SDO	received line	OPERATIONAL or PREOP	received line
Client SDO	one line	OPERATIONAL or PREOP	
Flying Master	active line	except force ResetComm, id master	
	received line	only id master response	
	both	only force **ResetComm**	
Redundancy	active		

Table 31, Line Dependent Message Transmission

4.13.3. Transmission of PDO

Transmission of PDOs is monitored by the producer to avoid transmission of too old messages (waiting too long for transmission.) This is done by the driver. Furthermore an error counter for the first Transmit PDO is managed. It will be incremented by 4 for each erroneous transmission and decremented by 1 for each error-free transmission. If the configured error limit (index $1F60_h$, sub-index 5) is reached the transmission of Heartbeat is stopped until the error counter is decremented to 0.

4.13.4. Indication Function

For each event detected by the redundant communication layer the user indication function *redundancyInd()* is called.

Values for Parameter *event*	Description
`REDUNCY_EVAL_TIMEOUT`	evaluation time is up
`REDUNCY_SWITCH_REDUNDANCY_LINE`	switch to redundant line
`REDUNCY_SWITCH_DEFAULT_LINE`	switch to default line
`REDUNCY_DEFAULT_LINE_OK`	default line ok
`REDUNCY_HB_ERROR`	default line Heartbeat failure
`REDUNCY_TPDO_FAILED`	TPDO error counter max value reached
`REDUNCY_TPDO_OK`	error counter decremented to 0

Table 32, Parameter Values for *redundancyInd*()

The default reaction of the CANopen Library can be averted by returning the value `CO_FALSE` when leaving the indication function. The following CANopen Library reactions are supposed:

Event	Return Value	Default Reaction of CANopen Libra
Eval-time is up	REDUNCY_EVAL_TIMEOUT	switch to redundant line
Switch to default line	REDUNCY_DEFAULT_LINE_OK	switch to default line
HB failure on default line	REDUNCY_HB_ERROR	switch to redundant line
PDO error counter reached	REDUNCY_TPDO_FAILED	switch off Heartbeat transmission

Table 33, Return Values for *redundancyInd*()

The redundancy communication can be tested using the examples **s11** (redundancy slave) and **m11** (redundancy master with Flying Master capabilities).

4.14. Nonvolatile Memory Usage

Every device needs some configuration data either for communication or application specific settings. This data has to be set at compilation time or after boot-up by a network configuration tool. It is substantially much more flexible to store configuration data in nonvolatile memory of the device. In order to do this the objects 1010_h and 1011_h are provided in the object dictionary. By writing a signature to these objects parts or the complete configuration data, as part of the object dictionary, can be stored in nonvolatile memory or restored from nonvolatile memory. Furthermore it is possible to load system values from ROM.

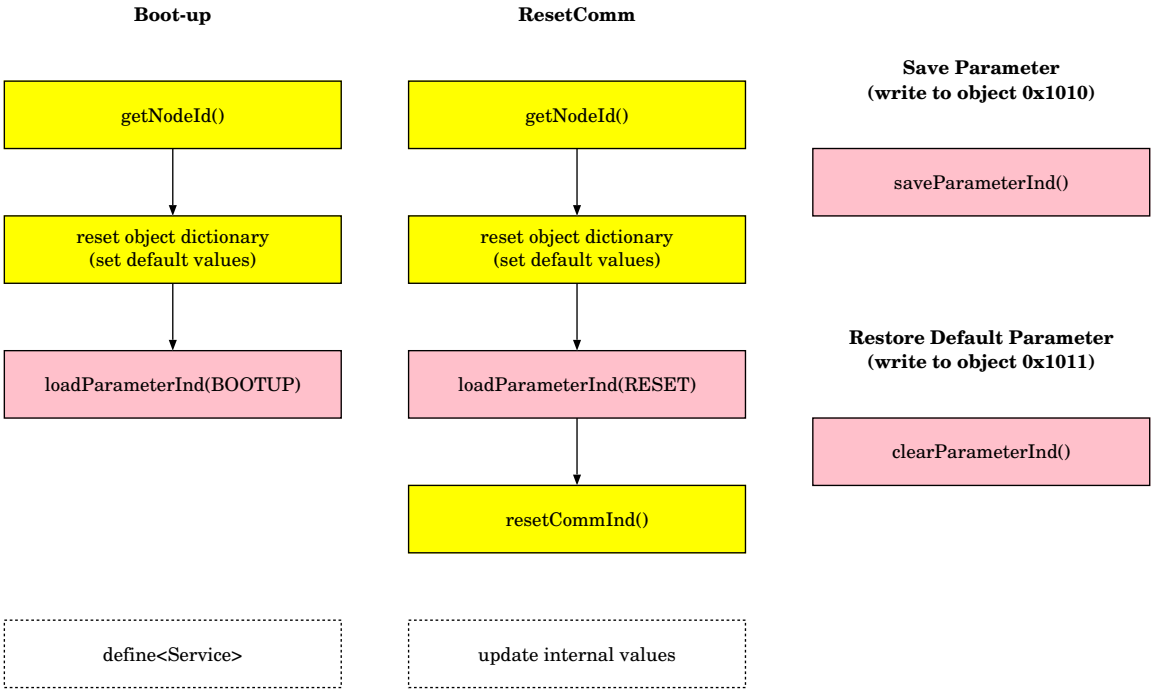


Figure 33, Restoring of Configuration Data after Reset

At power-on or after the NMT command *Reset Communication* all variable values in the object dictionary are reset to there *default*-value. This happens by restoring the variable entries of the object dictionary with values defined at compile time. Communication data

depending on the node-ID, like COB-IDs of PDOs, are calculated by the protocol CANopen Library according to there node-ID. In the next step the CANopen Library is calling the user function *loadParameterInd*(). The application has to read the stored data from non volatile memory and to restore corresponding object dictionary entries. The updated object dictionary data are then used to call the CANopen Library *define*-functions for the communication services (defineSdo(), ...) and the *update*-functions.

Data of the object dictionary can be written to non volatile memory via the object 1010_h. Writing the 4 byte signature *"save"* to this object causes the indication function *saveParameterInd*() to be called. With the provided parameter *sub-index* the user can select which data, which part of the object dictionary should be stored. It is up to the application programmer to choose which data to store.

Restoring configuration data happens with a write access to object 1011_h with the 4 byte signature *"load*". The re-loaded data becoming valid and visible in the object dictionary after one of the following NMT commands *Reset Communication* or *Reset Application* or a new boot-up. The CANopen Library is calling the user function *clearParameterInd*() after the object 1011_h was written. The application programmer is responsible to provide such code in this function that a following *loadParameterInd()* called after the NMT commands *Reset Communication* or *Reset Application* or after boot-up will not again overwrite the *default* values in the object dictionary with stored configuration data.

The simplest way doing that is erasing data previously stored in the non volatile memory area.

4.15. Layer Setting Services

CANopen addressing depends on a node-ID (1-127). Normally the node-ID is setup via DIP switches or rotary switches. Some devices can not provide DIP switches because they are completely sealed to be used in chemical applications or underwater. With the means of the Layer Setting Services CANopen devices can be identified and configured without external DIP switches. LSS services differentiate between and devices. In order to use LSS services it has to be initialized with a call to the function

```
defineLss(kind).
```

The function takes the mode (master/slave) as a parameter.

Within a CANopen network only one is allowed to exist. All other devices can be configured as s. All data for identifying a are taken of the identity object 1018_h. Every sub-index of object 1018_h has to be filled. The serial number has to be unique.

LSS distinguishes two sub-states:

- LSS WAITING mode
- LSS CONFIGURATION mode.

Switching between these states can be done globally for all nodes or selectively for just a single node. In the WAITING mode s only react upon the switch command and the *Identity Non Configured Mode* command. Within the CONFIGURATION mode all LSS commands can be used.

If the node does not have a valid node-ID (id = 255), then it transits automatically into the LSS WAITING state.

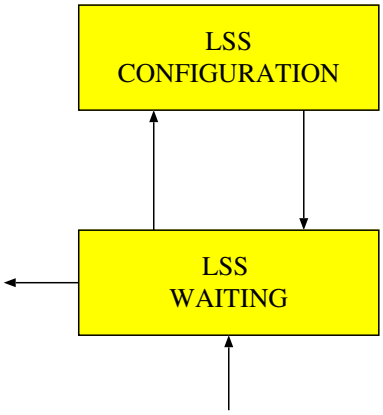


Figure 34, LSS Modes

Switching into the CONFIGURATION mode can be carried out independently of the current NMT state. If the node-ID was set and the device is set to WAITING mode again the CANopen Library will automatically call *Reset Communication* to verify the new COB-IDs for the *Predefined Connection Set.*

4.15.1. LSS Communication

Communication is always started by the , except for the *Identify Non Configured Mode* command, that can be issued by a , too, without order of the . This can happen when no valid node-ID was found. The sends its commands always with COB-ID 2021. After completion of a command the function

```
lssMasterCon(mode, parameter_1, parameter_2)
```

is called.
s always respond with COB-ID 2020. After reception of new parameters, like node-ID or bit rate, the function

```
lssSlaveInd(mode, parameter_1, parameter_2);
```

is called.

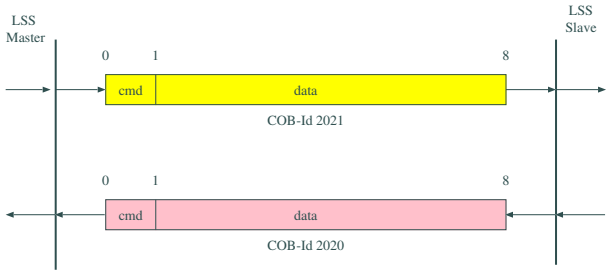


Figure 35, LSS Communication

LSS commands can be divided into three categories:

a) Switching between the sub-states

b) Configuration

c) Inquiry

4.15.1.1. Switching Between Sub-States

Globally switching from WAITING mode into CONFIGURATION mode is done unconfirmed with a single command. Switching a single node into the opposite mode requires sending the vendor-ID, product code, revision number and serial number of the desired device by the . If the slave exists in the CANopen network it responds to this request.

The programming interface for globally switching or selectively switching is the same.

```
lssSwitchMode(mode, vendor, product, revision, snr);
```

Parameter	Description
mode	`LSS_CONFIG_MODE, LSS_WAITING_MODE`
vendor	`0` - global, `!= 0` vendor-ID; 1018_h sub 1
product	product number; 1018_h sub 2
revision	revision number; 1018_h sub 3
snr	serial number; 1018_h sub 4

Table 34, Parameter *lssSwitchMode*

4.15.2. Configuration Services

LSS provides the configuration of the devices' node-ID and bit-timing without use of DIP switches or SDO transfer. The has to confirm the reception of the new parameter. On the the function

```
lssSlaveInd(mode, parameter_1, parameter_2);
```

is called on reception of a new parameter. Within this function the user can influence the return value within the LSS response.

The receives the response of the within the function

```
lssMasterCon(mode, parameter_1, parameter_2).
```

The new parameter have to be activated with another command by the .

Writing a new node-ID to a device is done with the function

```
writeLssConfigNodeIdReq(nodeId);
```

Prior to this the node has to be set into configuration mode with the function *writeLssSwitchModeReq()*. Activation of the new node-ID takes place during the transition from CONFIGURATION to WAITING mode. This is done with a call to the function *"Reset Communication"*.

Setting the bit-timings is done with a bit-timing table. This mechanism provides the possibility to use manufacturer specific timings. This bit-timing table has to be created by the programmer. The default bit-timing table of CANopen is supported.

```
writeLssConfigBitrateReq(table, tableIndex);
```

Parameter	Description
0	standard bit-timing table
1..127	reserved
128..255	manufacturer specific bit-timing table

Table 35, Parameter *writeLssConfigBitrateReq*

After changing the bit-timing it is only allowed to use LSS for *configuring the bit-timing, activate bit-timing* and *switch to another mode.*

Index	Bit Rate
0	1000 kbit/s
1	800 kbit/s
2	500 kbit/s
3	250 kbit/s
4	125 kbit/s
5	100 kbit/s
6	50 kbit/s
7	20 kbit/s
8	10 kbit/s

Table 36, Default Bit-Timings

After configuring a new bit-timing it can be activated with the function

```
activateNewBitrate(switchTime);
```

Activation of the new bit-timings is done in four steps:

1) empty send queue
2) wait switchDelay ms (sending not allowed)
3) switch to the new bit-timing
4) wait again switchDelay ms, before sending new messages

Saving of data in nonvolatile memory can be instructed via LSS.

```
writeLssStoreParameterReq()
```

On LSS slaves the steps for bit rate changing are indicated by the function

```
lssSlaveInd(mode, parameter_1, parameter_2);
```

mode	parameter 1	parameter 2	description
LSS_IND_BITRATE	table	index	new bit rate received
LSS_IND_BITRATE_SWITCH			switch bit rate command received
LSS_IND_BITRATE_SET			Timer 1 up - setup new bit rate
LSS_IND_BITRATE_ACTIVE			Timer 2 up - new bit rate can be used

4.15.3. Inquiry Services

Inquiry services are used to find nodes in a CANopen network or to select nodes. For this all data of the object 1018_h of the are requested. Every , that matches the requested data responds to the . With this method the can determine which slaves are connected to the network. If there is at least one node with the requested data, then the master is informed with the function *lssMasterCon*(). All data is requested with the same function.

```
writeLssInquiryReq(mode);
```

Mode	Description
LSS_VENDOR	search for vendor-ID
LSS_PRODUCT	search for product code
LSS_REVISION	search for revision number
LSS_SNR	search for serial number
LSS_NODEID	search for node-ID

Table 37, Parameter *writeLssInquiryReq* ()

The master can request all data of object 1018_h to identify a single node. By requesting particular ranges the exact data of nodes can be determined. With the function

```
lssIdentify(vendor, product, rev_low, rev_high, snr_low, snr_high);
```

a is able to scan the network and retrieve all connected s.

Paramter	Description
vendor	vendor-ID (from CiA)
product	product code
rev_low	revision number low
rev_high	revision number high
snr_low	serial number low
snr_high	serial number high

Table 38, Parameter *lssIndentify* ()

4.15.4. FastScan Service

FastScan service can reduce the time of a network scan for the following reasons:

- master sends only one instead of six remote identification messages
- it is possible to identify devices although skipping LSS numbers, e.g. the serial number may be ignored
- it is possible to identify vendor-id and product code

The fastScan service can be started by the LSS master by calling the function

```
writeLssFastScanReq(vendor, product, revision, snr);
```

The parameters vendor, product, revision and snr are 32 bit coded values, indicating the relevant bits for the fastScan compare with LSS slaves. All not used bits (reset bits) for vendor, product, revision and snr have to be zero at the LSS slaves identification object.

The fastScan service detects only one LSS slave for each cycle. It is always finished by the

```
lssMasterCon(mode, par1, par2)
```

function. If an unconfigured LSS slave was found, the

```
lssMasterCon(mode, par1, par)
```

is called with the parameter *LSS_CON_FAST_SCAN_DATA*. The parameter *par1* points to an array of 4 UNSIGNED32 values containing the identification of the found LSS slave. At this point the LSS slave is already in the LSS CONFIGURATION mode and can be setup with a new node id.

4.16. Safety with CANopen

Implementation of safety critical applications has to be carried out thoughtfully, because live of men and the protection of expensive goods and property depends on correctly-functioning software.

Therefore, these applications are subject to intense scrutiny by official inspection authorities.

This was the reason why the safety module was not fully embedded in the source code of the CANopen Library. It allows the user to program the main parts of the safety relevant parts of the implementation within the application.

4.16.1. Operation of Safety Critical Communication

For safety critical communication Safety Relevant Data Objects (SRDO) are used, that are mapped similar as PDOs. SRDOs are sent periodically. To increase safety, the data of an SRDO is transmitted twice. The second time the data is sent inverted. All SRDOs have their own COB-ID.

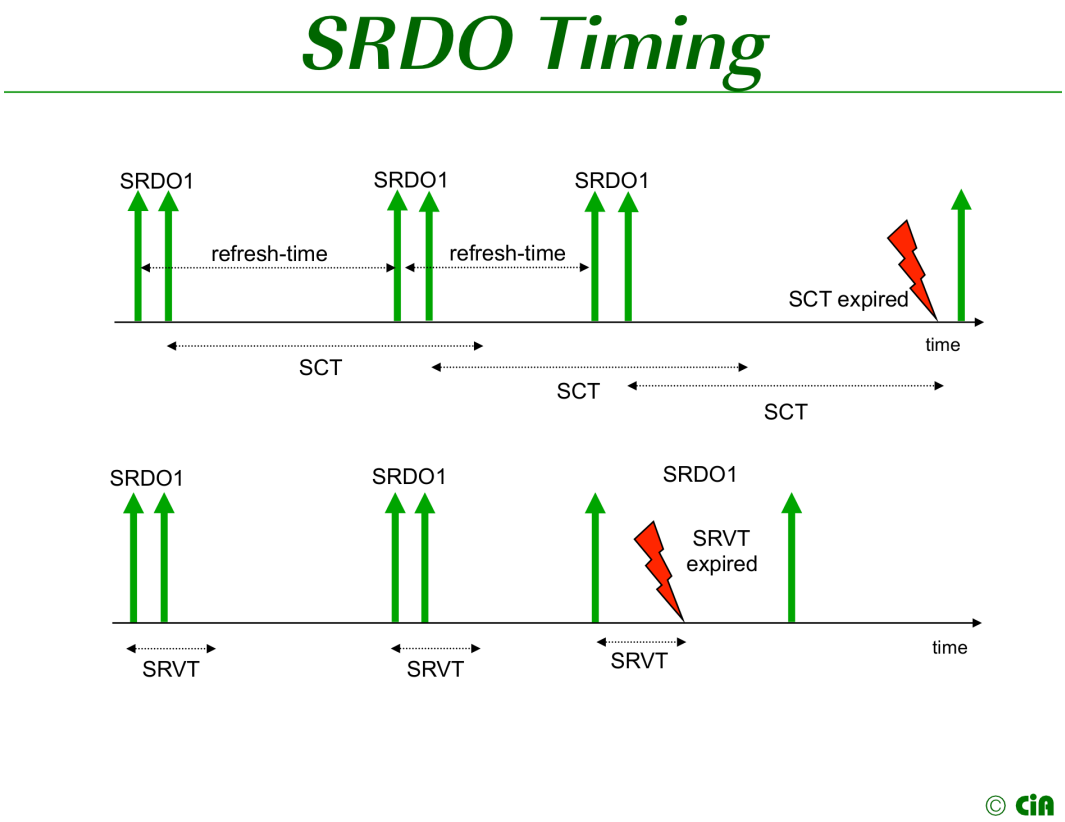


Figure 36, The Indication/Confirmation Interface
SCT Safeguard cycle time
SRVT - Safety relevant object validation time

The receiver of SRDO has to check the incoming SRDO for correctness of the order, correctness of data and the time period.

All information (timings, COB-ID, mapping) for SRDO are stored in the object dictionary.

4.16.2. Implementation

4.16.2.1. Object Dictionary

All SRDO communication parameters are stored in the object dictionary. The CANopen Library ensures that no safety relevant data is changed during OPERATIONAL state. Therefore access to this data is allowed only in state PRE-OPERATIONAL. Every access to SRDO communication data resets the configuration of SRDO. Then no SRDO communication is possible until the SRDO configuration is validated again.

Before SRDO parameters are valid, a checksum is calculated over the desired parameters of the object dictionary. This checksum is compared to the checksum stored in the object dictionary. If the two checksums are not equal, then SRDO communication is not allowed.

When changing mapping data of an SRDO the number of entries, i.e. sub-index 0, has to be set to 0. Consistency is checked when writing mapping data.

4.16.2.2. Initializing of SRDO

In order to use SRDOs they have to be initialized. With the function *defineSrdo*() the necessary internal structures and settings for the CAN-Controller are made.

4.16.2.3. Communication with SRDOs

Transmitting and receiving of SRDO is only possible in state OPERATIONAL. On transition to this state consistency is checked of the SRDO data of the object dictionary. If the configuration valid bit is not set SRDO are not enabled for sending or receiving. But the transition is still executed. If the configuration valid bit is set but there are inconsistencies transition to OPERATIONAL is aborted.

4.16.2.4. Transmitting SRDOs

Before sending SRDOs the CAN message has to be assembled according to the mapping. The user can use the function *mapSrdo*() to realize this.

When a SRDO should be sent the CANopen Library calls *mapSrdo*() and transmits the user assembled CAN message afterwards.

The function *writeSrdoReq*() sends an additional SRDO if it is necessary to do so.

4.16.2.5. Reception of SRDO

After reception of an SRDO the CANopen Library calls *srdoInd*() where the data can be processed. The user has to check for integrity of the data, i.e. comparison of not inverted and inverted data, and for the adherence of timing restrictions. The following has to be checked:

- CAN message in correct order

- adherence of the Safeguard cycle time (SCT)
- adherence of the Safety relevant object validation time (SRVT)

It is recommended to use a separate timer in order to realize the part of the safety completely in the application.

When all checks are done data can be saved into in the object dictionary.

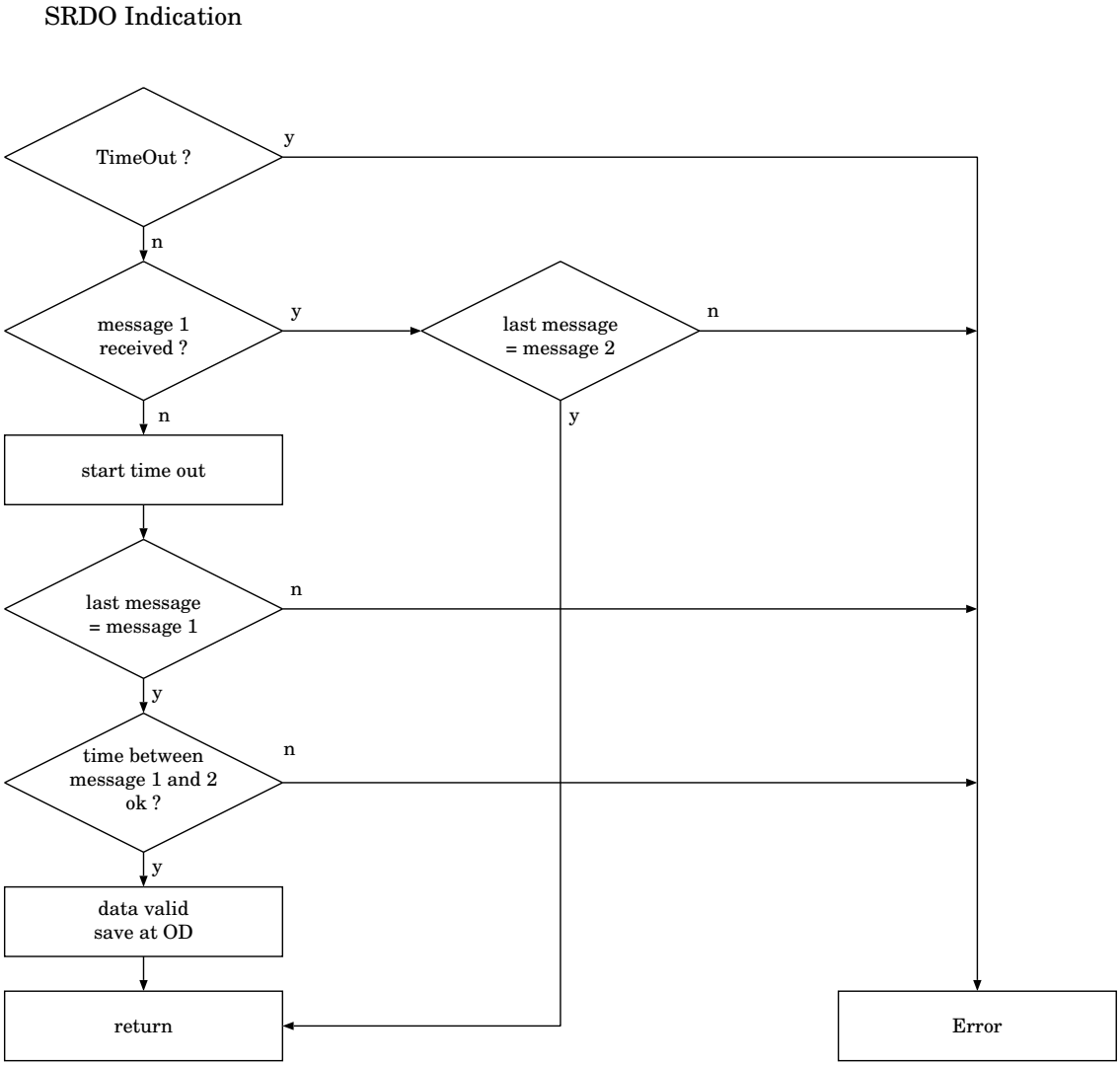


Figure 37, Flow Chart *srdoInd*()

4.16.2.6. Solution for SRDO Reception

Problems:

- execution time of indication function
- priority distribution on the bus (SRDO1 received, then reception of higher priority SRDO, then SRDO2)

- retain the driver concept without changes
- accurate timeout detection
- if many SRDOs (64) needed - i.e. 128 CAN messages

Possible solutions:

a) retain current concept and evaluation in the callback function

b) assign time stamps in ISR

c) usage of priorities within the buffer handling

d) calling of the callback function directly from the ISR

e) SRDO is valid after 1st and 2nd CAN message was received, i.e. timeout timer has to be reset only after the inverted SRDO was received.

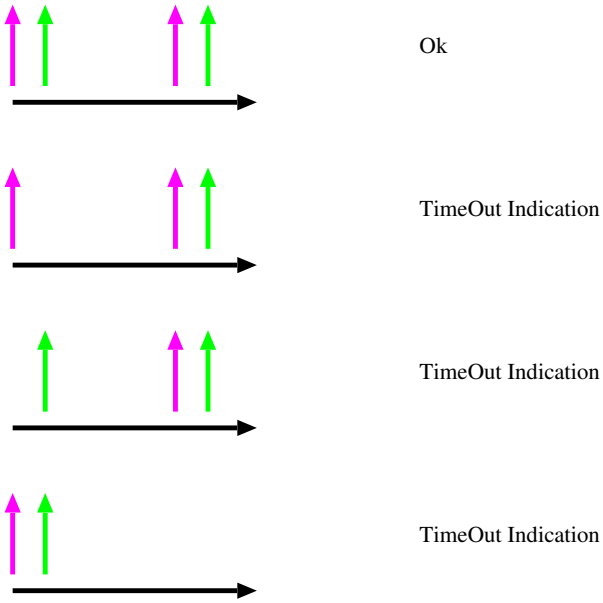


Figure 38, Timeout Causes of SRDO

4.17. LED Usage Conforming to CANopen

The CiA-303-3 provides a standardized way for state indication of a CANopen device. There is an error LED and a run LED. It is also possible to use only one of the two LEDs or instead of two single color LEDs a bicolor LED.
The run LED is green and indicates the CANopen state. The error LED is red and shows errors of the physical layer.

ERR LED	State	Description	Category
off	no error	The device is in working condition.	Mandatory
flickering	Autobaud/LSS	Auto baud rate detection or LSS services in progress	Optional
single flash	warning limit reached	At least one of the error counters of the CAN controller has reached or exceeded the warning limit.	Mandatory (*)
double flash	Error Control Event	A guard event (NMT slave or NMT master) or a Heartbeat event has occurred.	Mandatory
triple flash	Sync Error	The SYNC message has not been received within then configured communication cycle period time out (see index 1006_h).	Conditional (**)
Quadruple Flash	Event-timer error	An expected PDO has not been received before the event-timer elapsed.	Optional (**)
on	Bus Off	The CAN controller is bus-off.	Mandatory

Table 39, States Indicated by the ERR LED

(*) Should be optional, if there are CAN controllers available which do not indicate the warning level.
(**) Not supported

RUN LED	State	Description	Category
Flickering	Autobaud/LSS	Auto baud rate detection or LSS services in progress	Optional
Single flash	STOPPED	The device is in STOPPED state.	Mandatory
Blinking	PRE-OPERATIONAL	The device is in PRE-OPERATIONAL state.	Mandatory
On	*[opState]	The device is in *[opState] state.	Mandatory

Table 40, States Indicated by the RUN LED

4.17.1. Implementation

The LED functionality was implemented in the CANopen Library module **led.c**. The calls internal functions on occurrence of an error or event. Switching the LED on or off has to be done in the indication function

```
ledInd (led, action)
```

in module **usr_303.c.**

The CANopen LED can be activated with the *CANopen Design Tool*. LED functionality is controlled with the compiler directives:

Define	Description
CONFIG_CO_LED	activate **led.c** module
CONFIG_CO_RUN_LED	RUN LED is present
CONFIG_CO_ERR_LED	ERR LED is present
CONFIG_CO_BOTH_LED	both LEDs are present
CONFIG_CO_BICOLOUR_LED	bicolor LED is present

Table 41, Compiler Directives for CANopen LED

Example:

```
#ifdef CONFIG_CO_LED
void ledInd (
        UNSIGNED8 led,          /**< which CANopen LED */
        UNSIGNED8 action        /**< turn LED on or off */
    )
{
    if (led == CO_ERR_LED) {
        if (action == CO_LED_ON) {
            /* switch Error LED on */
            P2.3 = 1;
        } else {
            /* switch Error LED off */
            P2.3 = 0;
        }
    }
    if (led == CO_RUN_LED) {
        if (action == CO_LED_ON) {
            /* switch Status LED on */
            P2.4 = 1;
        } else {
            /* switch Status LED off */
            P2.4 = 0;
        }
    }
}
#endif /* CONFIG_CO_LED */
```

Listing 28, Example *ledInd*()

4.18. Virtual Objects

Normally all data have to be stored in the object dictionary of a device. However, for special applications it may be necessary to support additional or temporarily available objects besides the real objects. This objects are called virtual objects. Virtual objects are entries in the object dictionary that are managed by the user and have no physical entry in the object dictionary, i.e. have not been created with the *CANopen Design Tool*. They are placed in the manufacturer segment of the object dictionary and can only be accessed via SDO. Within the manufacturer segment virtual and real objects can be created in any order. A virtual object can not be appended to a real object.

The user is responsible for checking the data, value ranges and providing the necessary memory space.

In order to use virtual objects the compiler directive `CONFIG_VIRTUAL_OBJECTS` has to be set. Access to real objects in the object dictionary is done via SDO access and follows the attributes specified. If an object can not be found in the object dictionary access to a virtual object is assumed and the function *getVirtualObjAddr*() is called.

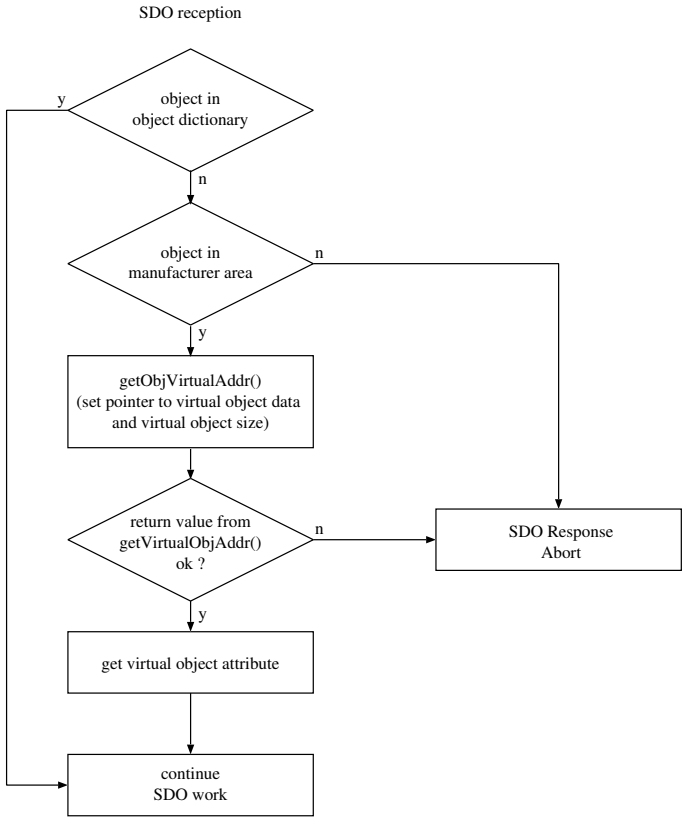


Figure 39, Flow Chart of an SDO Read Access to a Virtual Object

The application decides whether access to the virtual object with the given index and sub-index is permitted. If access is allowed then the pointer to the data and the size of the virtual object has to be provided by the application. If access is denied then an SDO abort is generated depending on the return value of the function. Further processing takes place in the same manner as for real objects.

It is assumed that virtual objects always are numerical and have read/write access.

4.18.1. Flow Chart for SDO Write Access

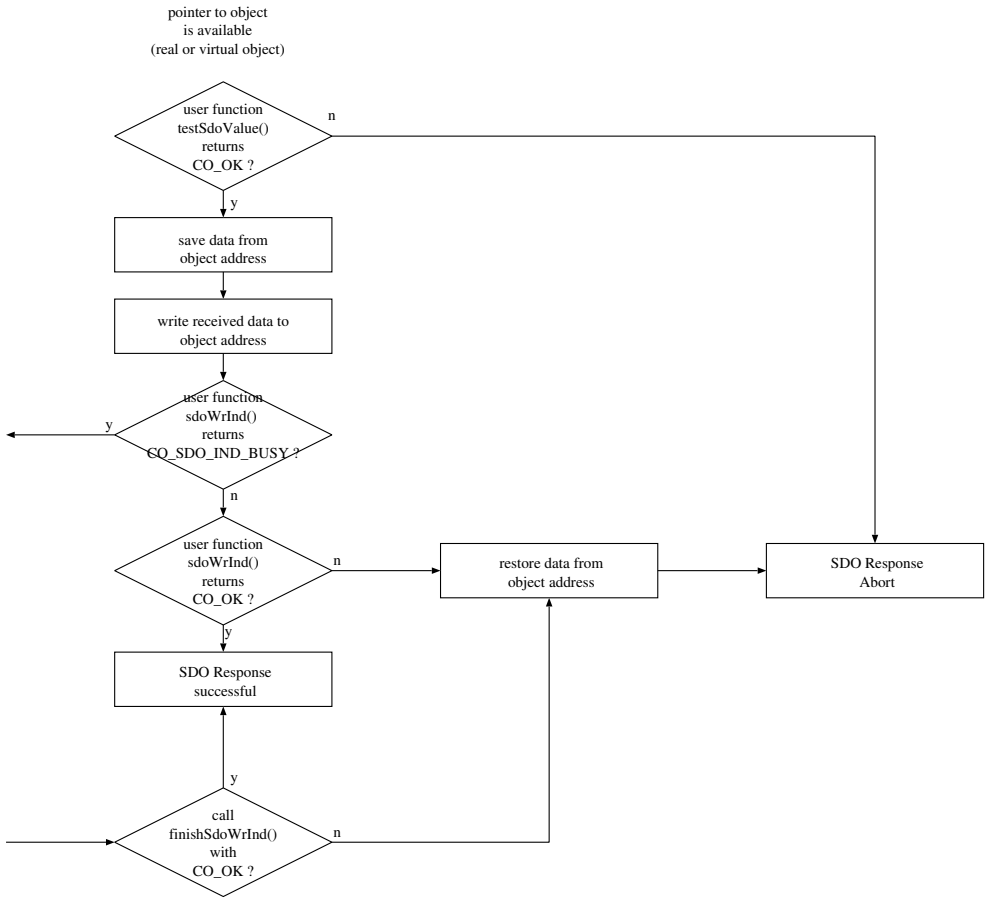


Figure 40, Flow Chart of an SDO Write Access to a Virtual Object

- CANopen Library checks for a virtual object. In case it is a virtual object then the function *RET_T getVirtualObjAddr(index, subIndex, *pData, *size)* is called. This function provides the pointer and the size of the object.
  At the same time an error return code can be passed back.
- Check if the given size matches the size given with SDO.
- *testSdoValue(index, subIndex, void *data, U32 size)* passes the data received.
  The user processes the data. The return value is either `CO_OK` or a defined SDO abort value.
- The CANopen Library writes the data at the address defined by *getVirtualObjAddr*()
- *writeSdoInd*() works the same way as for real objects.

4.18.2. Flow Chart for SDO Read Access

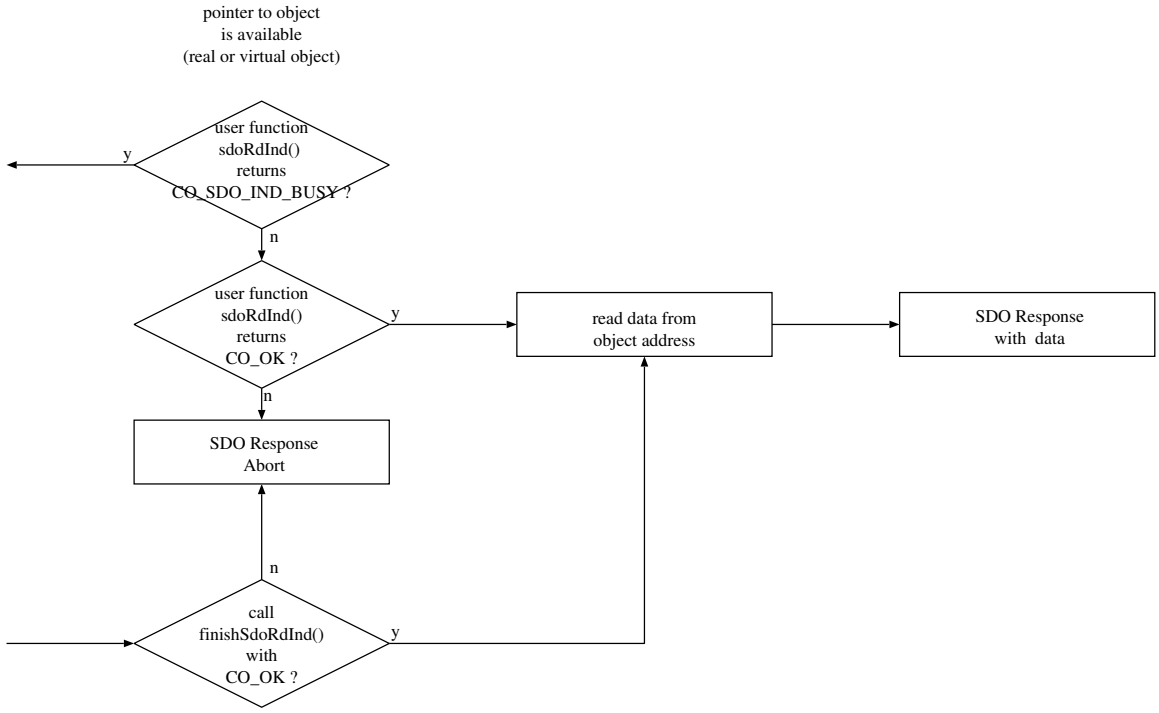


Figure 41, Flow Chart of an SDO Read Access to a Virtual Object

- CANopen Library checks for a virtual object. In case it is a virtual object then the function *RET_T getVirtualObjAddr(index, subIndex, *pData, *size)* is called.
- *readSdoInd*() works the same way as for real objects.
- Finally data read from the address provided by *getVirtualObjAddr*() and transmitted.

4.18.3. User-Functions

4.18.3.1. getVirtualObjAddr

*RET_T getVirtualObjAddr(U16 index, U8 subIndex, void *pData, U8 *size)*

This function is provided in the module **usr_301.c** and has to be filled by the user. It provides the CANopen Library with a correct pointer and size of the data. It is used for read and write access of a virtual object.

4.19. Object Callbacks

Sometimes an application has to react on an access of an object directly, no matter if the access was due to an PDO or an SDO or an other service. For such application behaviors the CANopen Library gained the possibility to attach application specific callbacks to an object. If the CANopen Library is configured for such a behavior the CANopen Library will call this callback.

4.19.1. Object Callbacks Function Pointer

The object specific callback function has the type CO_OBJ_CB_T , which is defined in co_acces.h.

```
typedef RET_T (*CO_OBJ_CB_T)(UNSIGNED16 /*index*/,
                             UNSIGNED8  /*subIndex*/,
                             CO_OBJ_CB_TYPE_T
                             CO_COMMA_LINE_PARA_DECL);
```

So if the user wants to use its own callbacks, the prototype of the function has to look something like this:

```
RET_T foo( UNSIGNED16 index, UNSIGNED8 subIndex,
           CO_OBJ_CB_TYPE_T reason CO_COMMA_LINE_PARA_DECL );
```

4.19.2. Object Callbacks Configuration

To globally enable the object callbacks the define CO_CONFIG_ENABLE_OBJ_CALL-BACK has to be set. This enables the code needed in the internal data structures and in the object entry structure. With the following defines the events can be configured on which the callbacks are called.

Define	Description
CO_CONFIG_OBJ_CB_PRE_PDO_READ	The callback is called before an object is read by an PDO.
CO_CONFIG_OBJ_CB_POST_PDO_READ	The callback is called after an object is read by an PDO.
CO_CONFIG_OBJ_CB_PRE_PDO_WRITE	The callback is called before an object is written by an PDO.
CO_CONFIG_OBJ_CB_POST_PDO_WRITE	The callback is called after an object is written by an PDO.
CO_CONFIG_OBJ_CB_PRE_SDO_READ	The callback is called before an object is read by an SDO.
CO_CONFIG_OBJ_CB_POST_SDO_READ	The callback is called after an object is read by an SDO.
CO_CONFIG_OBJ_CB_PRE_SDO_WRITE	The callback is called before an object is written by an SDO.
CO_CONFIG_OBJ_CB_POST_SDO_WRITE	The callback is called after an object is written by an SDO.

Table 42, Compiler directives for object callbacks

4.19.3. Object Callbacks Usage

If an callback is called the CANopen Library passes an parameter to the callback. With this parameter the callback knows the service number and the reason for the call. The type of this parameter is `CO_OBJ_CB_TYPE_T` and is descripted below.

```
typedef struct
{
    UNSIGNED16 reason;
    UNSIGNED16 serviceNbr;
}CO_OBJ_CB_TYPE_T;
```

The member `serviceNbr` contains the service number e.g. PDO number or SDO number. The member `reason` contains the reason why this callback is called. It could have the following states:

```
CO_OBJ_CB_TYPE_PRE_SDO_READ
CO_OBJ_CB_TYPE_POST_SDO_READ
CO_OBJ_CB_TYPE_PRE_SDO_WRITE
CO_OBJ_CB_TYPE_POST_SDO_WRITE
CO_OBJ_CB_TYPE_PRE_PDO_READ
CO_OBJ_CB_TYPE_POST_PDO_READ
CO_OBJ_CB_TYPE_PRE_PDO_WRITE
CO_OBJ_CB_TYPE_POST_PDO_WRITE
```

If the application needs to set or reset an object callback at runtime the function `setObjFuncPtr` can be used. The prototype of the function is descripted below.

```
RET_T setObjFuncPtr( UNSIGNED16 index, CO_OBJ_CB_T pNewFunc
                     CO_COMMA_LINE_PARA_DECL);
```

If the application wants to disable this callback, the new function pointer should be set to `NULL`.

5. Driver Interface

This chapter describes the driver interface of the CANopen Library. It shows how to build one's own driver. The CANopen Library Target Driver Interface consists of two modules. These are a CAN driver (**can<x>.c**) and a CPU- or RTOS driver (**cpu<y>.c**). A driver module is necessary for using the CANopen Library. The user is responsible for

these modules and thus they are always part of the user application. They have to be adapted for the application hardware. *port* provides many drivers for different targets. These drivers cover a wide range of systems, but they are not always the optimal solution for your application.

The prepared drivers work together with the examples of the CANopen Library. In order to ensure this fact, the engineers of *port* have inserted a layer between the actual driver and the CANopen Library.

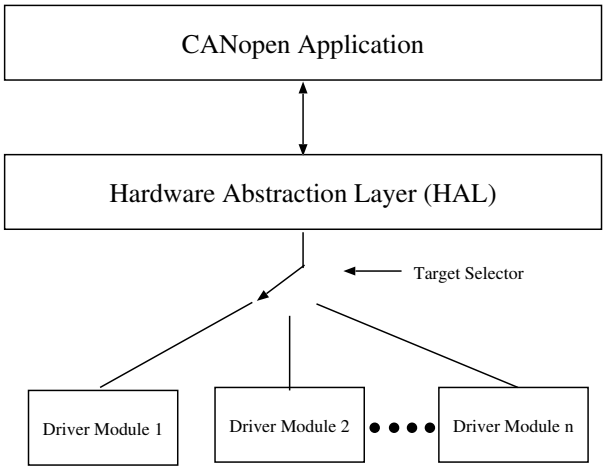


Figure 42, Hardware Abstraction Layer Principle

The hardware abstraction layer (HAL) realization is shown in figure 43. The example module **main.c** calls the HAL functions e.g. *initCAN*(). These functions are defined in the target specific initialization file i.e. **init_xc164.c**. The functions from this file call the functions from the driver modules i.e. **can_twincan.c**. The driver module uses a general include file i.e. **can_twincan.h**, which contains all CAN controller specific constants.
The module **examples.h** which is delivered with the CANopen Library, is only necessary to compile the examples from the delivery. It is not necessary for the end user project, although some parts may be useful.

If a customized driver module is not available, the user has the option to design it himself. A generic driver is available for a quick start. In this way the user can tailor the functions to his specific hardware properties[8].

[8] Please have a look at the next chapters for programming one's own drivers.

Figure 43, Hardware Abstraction Layer Realization

The HAL is not necessary when only one target is supported by the user application. In this case the driver functions can be called directly from the user's main routine. The driver modules used should be organized in the following manner (figure 44).

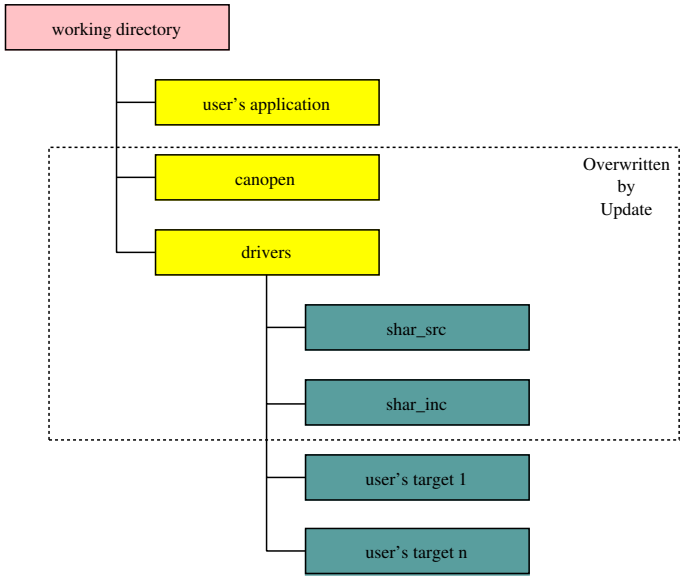


Figure 44, User's Driver Structure

A directory structure like this ensures that your driver will not be overwritten if a CANopen Library update is installed. Additionally the user is able to make the adaptations, which fit his needs best e.g. removal of unused functionality.

The following points give an overview of the driver requirements.

Figure 45 shows several CAN events to which the drivers has to react.

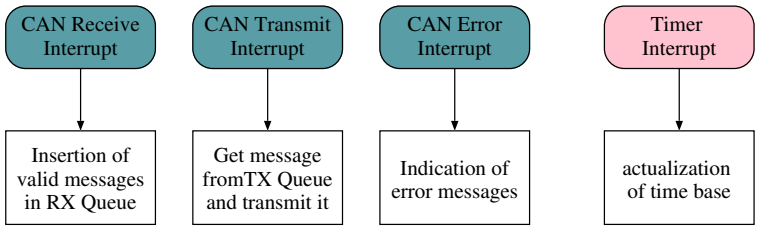


Figure 45, CANopen Driver Reaction

The driver has to process receive, transmit and error events on the CAN bus. These events can be initiated by the CAN controller or system messages from an operating system driver. The timer interrupt is responsible for updating the internal time base. It is used for all time based services, like Heartbeat, Node Guarding, synchronous services, for inhibit time and timeout checks. The reaction to the events mentioned above are shown in figure 45. Each event sets a global flag when it is triggered. These flags are evaluated by the CANopen Library. No CANopen Library function is called directly by the driver. This means the CANopen Library has to be called cyclically in the main loop.

➥ All driver functions are part of the user application. Only a few functions are called directly by the CANopen Library. The interface between the CANopen Library and the driver can not be changed. Its functions are described in the following paragraphs.

Notes for development of a driver

The driver is composed of the following parts:

1) compiler adaptations/ CPU dependent functions
2) CAN controller specific functions
3) application dependent adaptations

5.1. CAN Driver

5.1.1. Prepared CAN Driver

The prepared driver structure is shown in figure 46. It consists of a hardware independent and a hardware dependent part. Hardware independent are the receive (RX) and transmit (TX) buffers and their access mechanism. All of these functions are coded in **drivers/shar_src/cdriver.c**.

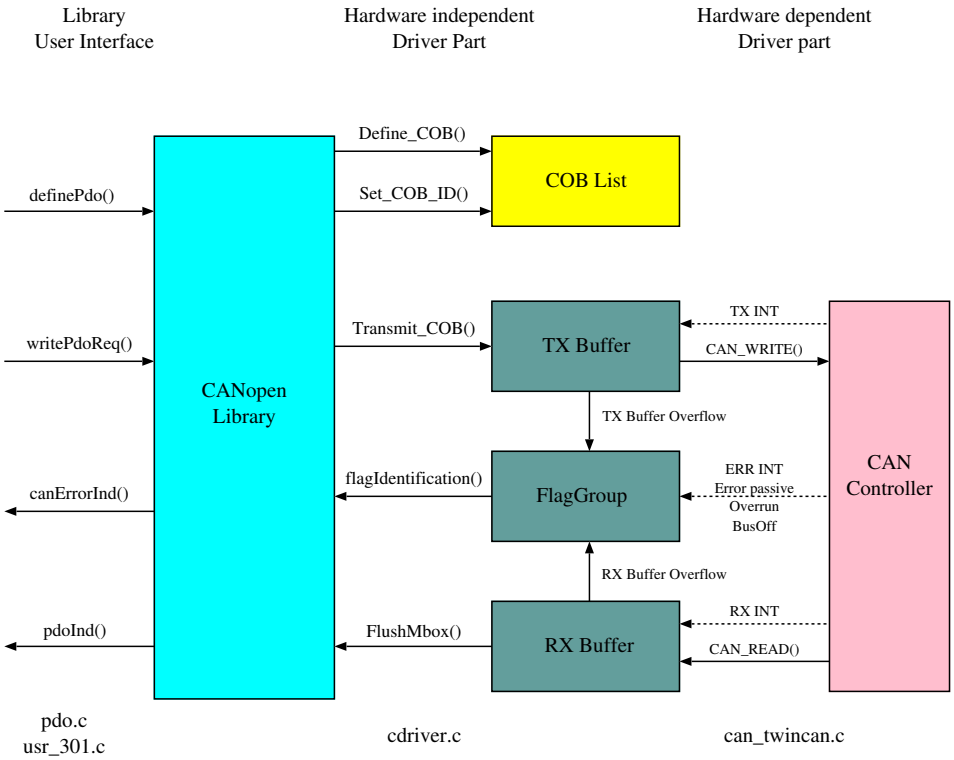


Figure 46, Prepared Driver Structure

5.1.2. CANopen Driver API

Initialization

Init_CAN()
The parameters of the function *Init_CAN*() are depending on the CAN controller and its hardware access. The CAN controller is initialized but left in the state stopped.

getBitTiming()
Returns a pointer to an entry of the bit-timing table for the bit rate that was passed as argument.

Set_Baudrate()
Initializes the CAN controller for the given bit rate. The CAN controller is left the state stopped.

CAN Bus Status

Start_CAN()
Turns the CAN controller from state stopped into state running and enabled the CAN interrupt.

Stop_CAN()
Turns the CAN controller into state stopped.

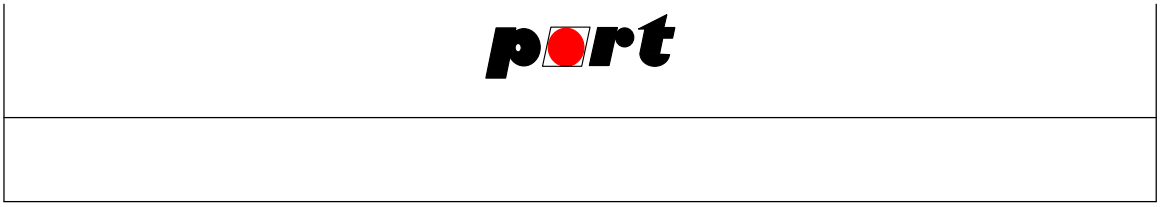
Clear_busoff()
After a bus-off this function turns the CAN controller into state running. If this is successful is not sure. The state change can last some time (128x11 recessive Bits).

Communication Objects (COB)

Communication objects are used for administration of static information of CAN messages. They are of data type `COB_T` (see chapter "CAN Driver Basics").

Define_COB()
A new communication object is created. In the TwinCAN driver the helper function *createCob*() takes over the biggest part of the functionality. For Full-CAN controller like the TwinCAN hardware message objects are initialized, too. This is mainly done in the function *initChannel*().

Set_COB_ID()
Sets a new COB-ID for an existing communication object. The type (RX, TX) of the communication object can change. It might as well be deactivated. Setting the COB-ID in a hardware message object in the TwinCAN driver is mostly done in the function *initChannel*().



Transmit_COB()
Sends a communication object. In drivers for embedded devices the message is written into the software buffer in the function *Insert_TX_Request*(). The function *GetNext_TX_Request*() activates sending the message.

CAN Driver State

getCanDriverState()
Returns the current state of the CAN driver. (e.g. `CANFLAG_ACTIVE`, `CANFLAG_BUSOFF`).

Interrupts

CAN_int()
This is the interrupt function. In many drivers all interrupts are handled in just one function.

➥ Please have a look at the Reference Manual for detailed function descriptions.

5.1.3. CAN Driver Basics

The CAN driver is the interface for reception and transmission of messages. In general message queues are used for all messages received (`pRX_Buffer[]`) and transmitted (`pTX_Buffer[]`).
For error messages and timer events a group of global flags (`coLibFlags`, `coCanFlags`) are defined. The flags can be set by the CAN and the timer interrupt. In the function *flagIdentification*() the flags are evaluated. It is called within the function *FlushMbox*() located in the module **cdriver.c** or in a cpu specific module.

Received messages are processed by the function *msgIdentification*(). The argument passed to this function is of the type `CAN_MSG_T`. It is defined in the module **co_stru.h** and contains the CAN message. Assembling the CAN message into a variable of type `CAN_MSG_T` and the call to *msgIdentification*() is done in the function *FlushMbox*() from the module **cdriver.c**.

```
struct CAN_MSG
{
    COB_KIND_T  cobType;    /* COB Type */
    COB_IDENT_T cobId;      /* COB Id */
    UNSIGNED8   pData[8];   /* data */
    UNSIGNED8   length;     /* if bit CO_RTR_REQ
                             * is set -> RTR */
};

typedef struct  CAN_MSG         CAN_MSG_T;
```

Listing 29, CAN Message Structure

Transmission of CAN messages is done in the function *Transmit_COB*(). This function is defined in the module **can_xxx.c** of the CAN driver. The parameters of this function are a pointer to the COB type description structure (`COB_T`), a pointer to the data, and the CAN line number for the multi-line version. The most important elements of the structure `COB_T` are shown in table 43.

Type	Name of Element	Remarks
`COB_IDENT_T`	`cobId`	ID of the message
`COB_KIND_T`	`eType`	type of message, see table 44
`UNSIGNED8`	`bChannel`	channel number (CAN driver internal)

Table 43, *COB_T* Members

The type *COB_KIND_T* stores the information about the use of the message object, i.e. receiving, sending, receiving RTR or sending RTR messages. This is the main task. In addition the CANopen service like PDO, SDO can be specified too.

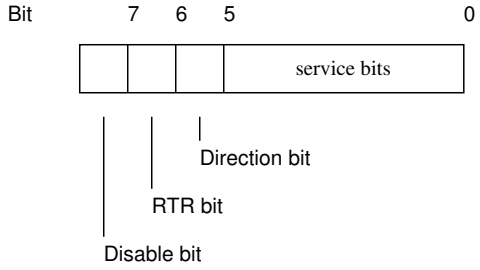


Figure 47,

➥ Note: The concrete position of the bits are subject to change. Only use the symbolic names of the CANopen Library.

Type	Remarks
`CO_COB_RX`	receive message
`CO_COB_TX`	transmit message
`CO_COB_RX_RTR`	receive message, which can be requested by the local device (a receive message that can send an RTR request)
`CO_COB_TX_RTR`	transmit message, which can be requested by remote devices (a transmit message, which can receive an RTR request)

Table 44, Type of CAN Message

CAN error handling covers hardware and software errors, like CAN passive or buffer overflow. All CAN errors are reported by setting a flag in the global variable *coLibFlags*. The concrete error cause is set in the global variable *coCanFlags*. The function *flagIdentification*() checks the variables *coLibFlags* and *coCanFlags* and calls the appropriate CANopen Library function or a user indication function.

The definition of the CAN error flags and the timer flags is shown in table 46.

Flag Name(coLibFlags)	Specification
COFLAG_SYNC_RECEIVED	Sync message was received
COFLAG_TIMER_PULSED	timer interval has pulsed
COFLAG_CAN_EVENT	CAN event

Table 45, CAN Error and Timer flags part 1

Flag Name(coCanFlags)	Specification
CANFLAG_INIT	CAN is in INIT state
CANFLAG_ACTIVE	CAN is active
CANFLAG_BUSOFF	CAN controller is in bus-off
CANFLAG_PASSIVE	CAN controller is in error passive
CANFLAG_OVERFLOW	CAN controller has detected an overflow
CANFLAG_TXBUFFER_OVERFLOW	TX Buffer overflow at the CAN driver
CANFLAG_RXBUFFER_OVERFLOW	RX Buffer overflow at the CAN driver

Table 46, CAN Error flags part 2

5.1.3.1. Adaptation of the flag handling

The default implementation of the flag set/reset macros are located in **co_flag.h** and **co_drv.h**.

```
#define SET_COLIB_FLAG(FLAG)   (GL_ARRAY(coLibFlags) |= (FLAG))
#define RESET_COLIB_FLAG(FLAG) (GL_ARRAY(coLibFlags) &= ~(FLAG))
#define SET_CAN_FLAG(FLAG)    GL_ARRAY(coCanFlags) |= (FLAG)
#define RESET_CAN_FLAG(FLAG)  GL_ARRAY(coCanFlags) &= ~(FLAG)
```

Changes are possible by a user specific setting in the **cal_conf.h**. A good working change is an additional `atomic` command or anything alike.
An example change could be

```
#define SET_CAN_FLAG(FLAG)    do{\
    DISABLE_CPU_INTERRUPTS();\
    GL_ARRAY(coCanFlags) |= (FLAG);\
    RESTORE_CPU_INTERRUPTS();\
}while(0)
```

5.1.3.2. Adaptation of the FlushMbox() function

The default implementation of *FlushMbox*() is located in **cdriver.c**. For an own implementation this function can disabled by removing the define `CONFIG_COLIB_FLUSHMBOX`. With the *CANopen Design Tool* this is done by deselecting the hardware setting: *Driver uses CANopen Library function FlushMbox()*.

5.1.4. Buffer Handling in Embedded Drivers

Embedded CAN drivers use separate buffers for sending and receiving CAN messages. Sending and receiving is carried out interrupt driven. For drivers used in an operating system environment like Windows or Linux buffer handling is done by the layer 2 driver of the system. This driver is provided by the CAN interface manufacturer.

The default buffer handling is activated in the *CANopen Design Tool* with the option *"Driver uses CANopen Library buffer"*. The size of the send and receive buffer can be set independently of each other. The *CANopen Design Tool* generates the following settings in the header file **cal_conf.h:**

```
#define CONFIG_COLIB_BUFFER    1
#define CONFIG_TX_BUFFER_SIZE  10
#define CONFIG_RX_BUFFER_SIZE  10
```

This example uses 10 entries separately in the send and receive buffer. The data in the buffers are of type `BUFFER_ENTRY_T`.

```
typedef enum  { EMPTY, FULL } MEM_STAT_T;

struct BUFFER_ENTRY
{
        VOLATILE MEM_STAT_T     eStat;
        COB_KIND_T      eType;
        COB_IDENT_T     cobId;
        UNSIGNED8       bLength;
        UNSIGNED8       pData[8];
        UNSIGNED8       bChannel;
};

typedef struct BUFFER_ENTRY XDATA BUFFER_ENTRY_T;
```

Listing 30, CAN Buffer Entry Structure

Buffer handling for embedded drivers uses macros for read/write access. The macros are defined in the header file **cdriver.h..** The parameter for these macros are:

Parameter	Value	Remarks
direction	`TX`	transmit buffer
	`RX`	receive buffer
action	`Read`	read from the buffer
	`Write`	write to the buffer
source		`BUFFER_ENTRY_T` member
destination		`BUFFER_ENTRY_T` member

Table 47, Buffer Access Macro Parameter

For all macros it is assumed that a local pointer variable

```
BUFFER_ENTRY_T * pBuffer;
```

exists.

BUFFER_INIT_PTR(direction, action)

Initialize access to the buffer.

BUFFER_READ(direction,source)
Read from buffer

BUFFER_WRITE(direction,destination,data)
Write *data* to the buffer

BUFFER_ENTRY_INCR (direction,action, status)
Switch to next entry of the buffer. After a call to this macro the previous *BUFFER_INIT_PTR*() is invalid. To receive access to a buffer *BUFFER_INIT_PTR*() has to be called again.

CHECK_BUFFER_READ (direction)
Checks if the buffer contains a "full" entry and if a message can be read from this buffer. If the check is true the following code block is executed.

```
CHECK_BUFFER_READ(RX)
{
    /* read from the Buffer */
    ...
}
```

CHECK_BUFFER_WRITE (direction,error)
Checks if the buffer contains an "empty" entry and if a message can be written to this buffer. If the check is true the following code block is executed. Otherwise an *error condition* is signaled and the following code block is ignored.

```
CHECK_BUFFER_WRITE( TX, CANFLAG_TXBUFFER_OVERFLOW)
{
    /* write to the Buffer */
    ...
}
```

Process flow of a buffer read cycle

```
{
BUFFER_ENTRY_T * pBuffer;

    /* allow buffer access */
    BUFFER_INIT_PTR(TX, Read);

    /* buffer full? */
    CHECK_BUFFER_READ(RX)
    {
        /* read from the buffer */
        length = BUFFER_READ(RX, bLength);
        ...
        /* release buffer */
        BUFFER_ENTRY_INCR( RX , Read , EMPTY);
    }
}
```

Listing 31, Buffer Handling

In **cdriver.c** there are more functions that support this buffer handling.

void clearTxBuffer(void) *void clearRxBuffer(void)*
Marks all entries in one buffer RX/TX as empty. Messages that are in buffers of the CAN controller remain unchanged.

BUFFER_INDEX_T getNumberOfTxMessages(void) *BUFFER_INDEX_T getNumberOfRxMessages(void)*
Returns number of RX/TX messages in the buffer.

*RET_T Insert_TX_Request(COB_T * pCOB, UNSIGNED8 * pData)*
Inserts the next transmission request into the queue.

➥ Please have a look at the Reference Manual for detailed function descriptions.

5.1.5. Interrupt Handling

The interrupt handling is shown in figure 48.

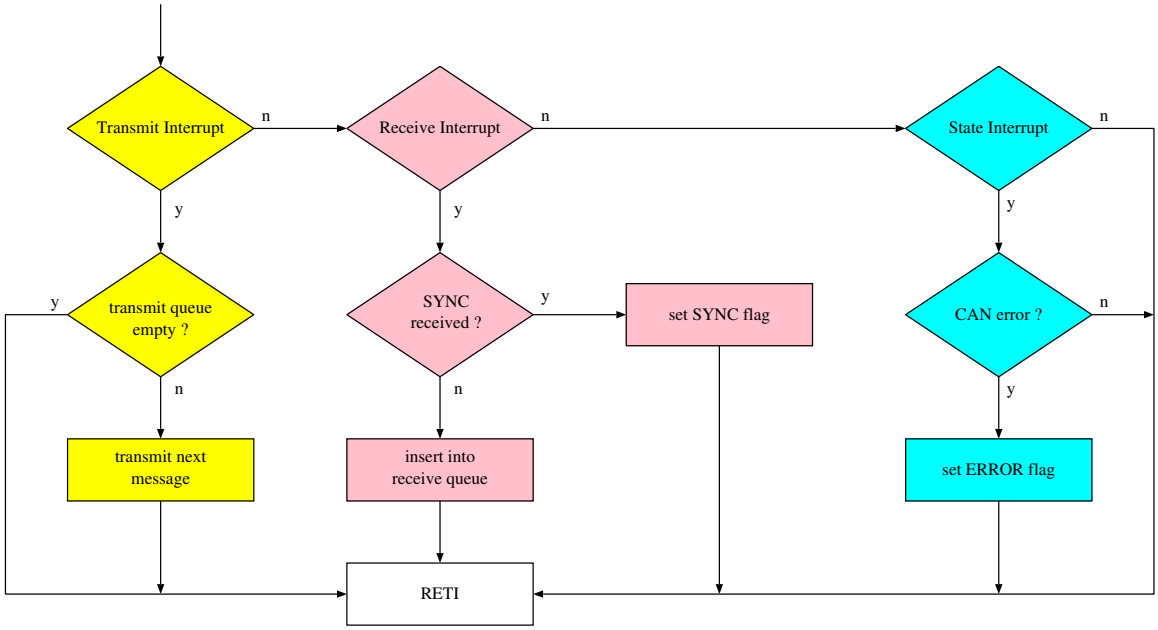


Figure 48, CAN Interrupt Handling

For systems which do not use a CAN interrupt e.g. PCs using active CAN cards or systems which get the CAN message via operating system drivers, it is not necessary to put the incoming messages into a queue, because all messages are typically buffered in the operating system driver or in the memory of active CAN cards. The interpretation can be done directly via *msgIdentification*() (see above). The reception flow checks two kinds of messages. The first is the CANopen SYNC telegram. The classification of SYNC has to be the first in order to guarantee a minimum of jitter. All other messages are put into a queue which decouples the CAN interrupt service routine from the CANopen Library.

CAN-ISR Management

- *SetIntMask*()
- *ResetIntMask*()
- *Init_CAN_Interrupts*()
- *Enable_CAN_Interrupts*()
- *Disable_CAN_Interrupts*()
- *Restore_CAN_Interrupts*()

For the CAN-ISR management the functions for enabling/disabling CAN interrupts are mandatory. These functions ensure that the values of the message queue read and write pointers remain consistent. For active CAN modules these functions are empty. The functions for the setting and resetting of the CAN-ISR to/from interrupt vector tables are not implemented for active modules.

5.1.6. Driver Example XC164

The embedded driver DP XC164CS consists of the

- CPU Driver Infineon XC166,
- CAN Driver Infineon TwinCAN

and an adaption for the used development board located in the directory *drivers/xc164/.*

Compiler header files exist for

- the Keil compiler *(drivers/shar_inc/co_keil.h)* and
- the Tasking compiler *(drivers/shar_inc/co_tasking.h).*

This combination of CPU and CAN controller is setup with the defines

```
#define CONFIG_CPU_FAMILY_XC166
#define CONFIG_CAN_FAMILY_TWINCAN
#define CONFIG_COMPILER_KEIL_C166
```

in the file **cal_conf.h**

5.1.6.1. Basics

The TwinCAN controller has 32 configurable hardware message objects. The number of hardware message objects allows to support 3 different software modes in the driver. The modes are enabled in the file **cal_conf.h.**

SoftwareModes

without define
The CAN controller is used in Basic-CAN mode. All messages are received. In this mode the number of CANopen services is only limited by RAM of the CPU.

CONFIG_CAN_FULLCAN_SOFT_RTR
The CAN controller is used in Full-CAN mode. The hardware filtering of the CAN controller is used. RTR messages are processed by the CANopen Library and not by the CAN controller. The number of CANopen services is limited.

CONFIG_CAN_FULLCAN_SOFT_RTR *CONFIG_CAN_ONLY_ONE_TRANSMIT_CHANNEL*
This is an extension of the Full-CAN mode. It increases the number of CANopen services for the Full-CAN mode. All send objects that do not use RTR use one hardware message object. To efficiently use this setting support for RTR should be avoided. This is achieved by setting bit 30 (`PDO_NO_RTR_ALLOWED_BIT`) in a COB-ID of a TPDO.

AccessModes

Two different modes for access to the CAN controller are supported.

CONFIG_CAN_ACCESS_MEM_MAP
The CAN controller is located in memory address range of the CPU. Access to the CAN controller can be carried out by pointer.

CONFIG_CAN_ACCESS_IO_MAP
The CAN controller is addressed via I/O functions. It is also possible to use SPI for access. Access by pointer is not allowed.

Register Layout

Register layout of the CAN controller can be defined with the following macros.

CONFIG_BIG_ENDIAN
Access with a Big Endian machine (CPU is Big Endian)

CONFIG_CAN_REGISTER_OFFSET
Address offset between to consecutive registers

Macros for Accessing the CAN Controller

Access to the CAN controller is carried out by a number of access macros. In access mode `CONFIG_CAN_ACCESS_MEM_MAP` the variable *addr* is a pointer to an address in the CAN controller. Unlike in the access mode `CONFIG_CAN_ACCESS_IO_MAP` the variable *addr* is a value of a data type, e.g. `UNSIGNED16`, that is supported by the I/O functionality. The use of macros for accessing the CAN controller simplifies adaption of a driver to a different CPU.

CAN_ADDR_T
Data type for accessing the CAN controller e.g. (UNSIGNED8 *)

void CAN_INIT_BASE_PTR(addr)
Initializes address pointer to the CAN controller to address of *addr*.

CAN_ADDR_T CAN_ADDR(reg)
Returns the absolute address of a register *reg*.

UNSIGNED8 CAN_READ_PTR(CAN_ADDR_T addr) *UNSIGNED16 CAN_READW_PTR(CAN_ADDR_T addr)* *UNSIGNED32 CAN_READL_PTR(CAN_ADDR_T addr)*
Reads a value of the absolute address *addr*. The address *addr* should be retrieved with the macro *CAN_ADDR*().

UNSIGNED8 CAN_READ(reg)
UNSIGNED16 CAN_READW(reg)
UNSIGNED32 CAN_READL(reg)

Reads the register *reg*. This is equivalent to: *CAN_READ_PTR(CAN_ADDR(reg))*

void CAN_WRITE_PTR(CAN_ADDR_T addr, UNSIGNED8 data) *void CAN_WRITEW_PTR(CAN_ADDR_T addr, UNSIGNED16 data)* *void CAN_WRITEL_PTR(CAN_ADDR_T addr, UNSIGNED32 data)*
Writes the value of *data* to the absolute address *addr* of the CAN controller. The address *addr* should be retrieved with the macro *CAN_ADDR*().

void CAN_WRITE(reg, UNSIGNED8 data) *void CAN_WRITEW(reg, UNSIGNED16 data)* *void CAN_WRITEL(reg, UNSIGNED32 data)*
Writes the value of *data* to the register *reg*. This is equivalent to: *CAN_WRITE_PTR(CAN_ADDR(reg)).*

void CAN_SET_BIT(reg, UNSIGNED8 bitfield) *void CAN_SET_BITW(reg, UNSIGNED16 bitfield)* *void CAN_SET_BITL(reg, UNSIGNED32 bitfield)*
Sets all bits of *bitfield* to register *reg*.

void CAN_RESET_BIT(reg, UNSIGNED8 bitfield) *void CAN_RESET_BITW(reg, UNSIGNED16 bitfield)* *void CAN_RESET_BITL(reg, UNSIGNED32 bitfield)*
Clears all bits of *bitfield* to register *reg*.

UNSIGNED8 CAN_TEST_BIT(reg, UNSIGNED8 mask) *UNSIGNED16 CAN_TEST_BITW(reg, UNSIGNED16 mask)* *UNSIGNED32 CAN_TEST_BITL(reg, UNSIGNED32 mask)*
Reads register *reg* and applies *mask* to the value.

➥ Note: The drivers support only one access method to the CAN controller like byte, word or long access.

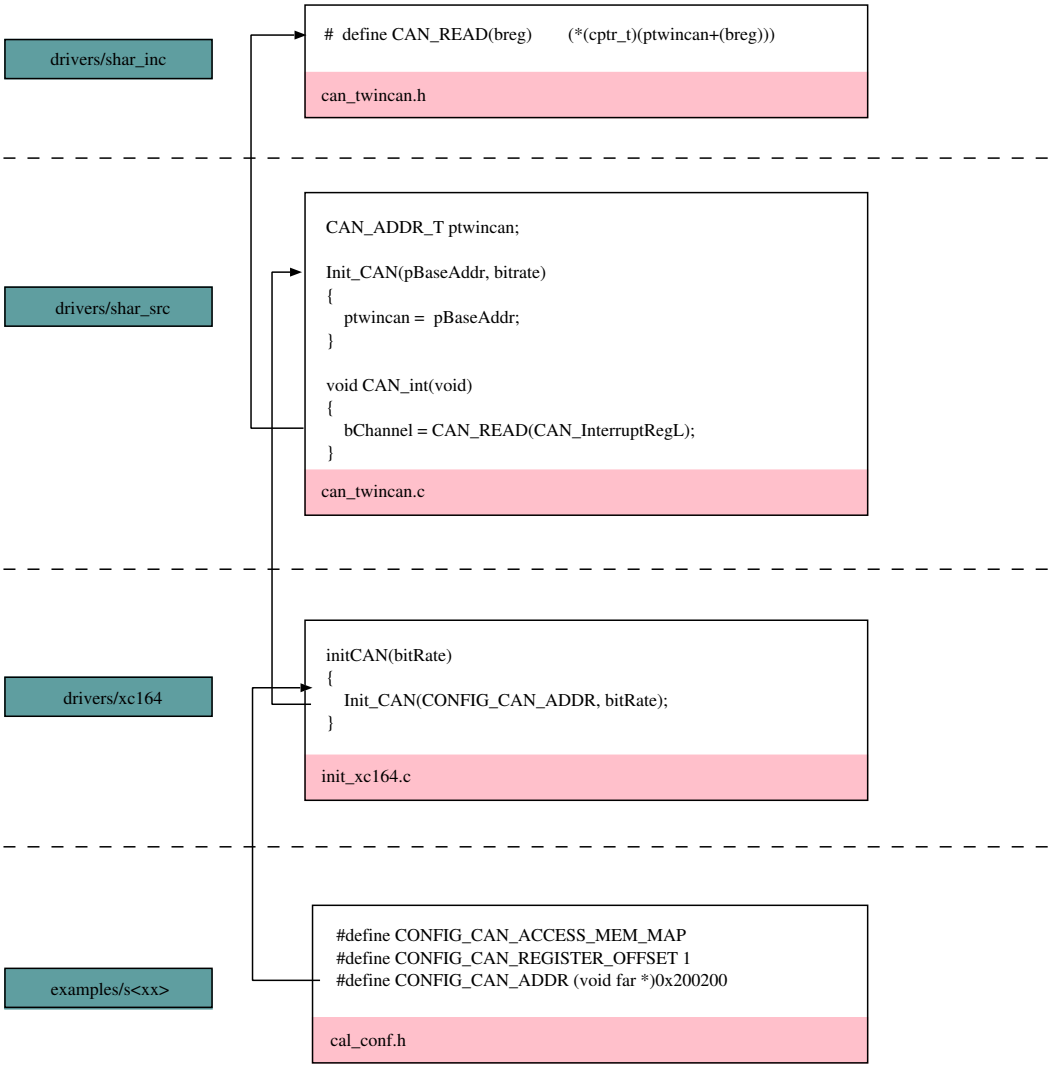


Figure 49, CAN Driver Macros

Macros for Accessing Hardware Message Objects

Specifically for **Full-CAN Controller** there are macros for accessing hardware message objects directly that are based on the above mentioned macros. An adaption for prepared drivers is normally not necessary. To use these macros the local variable

```
CAN_ADDR_T pChannel;
```

is needed.

CAN_INIT_OBJ_PTR(channel)
Access to hardware message object *channel* of the CAN controller. This macro has to be invoked every time before a hardware message object is accessed.

CAN_READ_OBJ(channel, reg)
Read register *reg* of hardware message buffer *channel*.

CAN_WRITE_OBJ(channel, reg, data)
Write the value of *data* to register *reg* of the hardware message object *channel*.

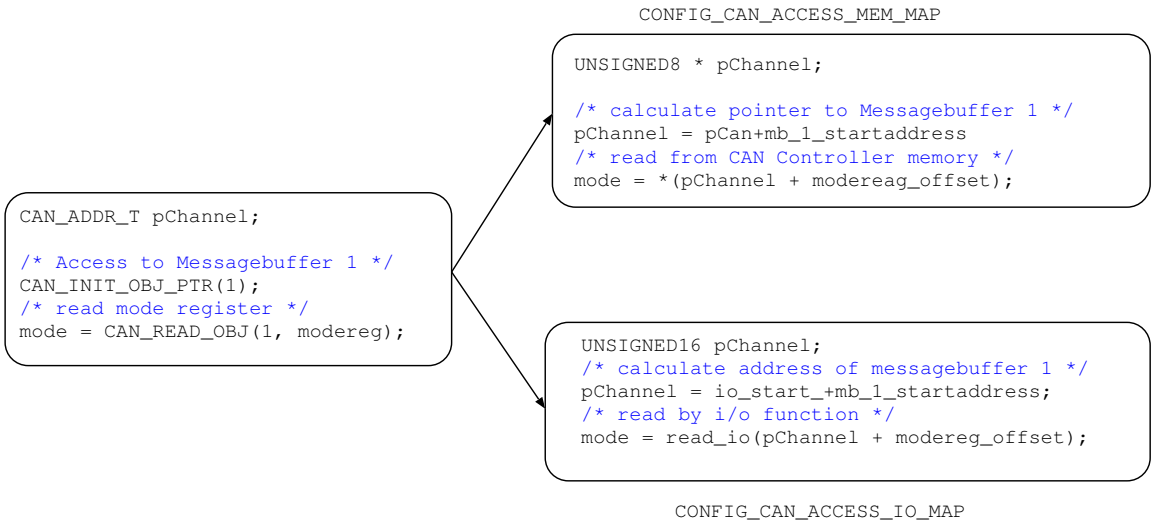


Figure 50, Driver Macros Full-CAN Controller

5.1.6.2. Bit-timing Table

The bit-timing table of the driver has to be customized to the used hardware. This is necessary because the values in this table depend on the CAN clock of the CPU. For some preselected CAN clocks the table is filled with values. The bit-timing tables are located in the header file of the CAN driver in question.
For the TwinCAN driver they are defined in **drivers/shar_inc/can_twincan.h**. The selection of the appropriate table is done in the hardware settings menu of the *CANopen Design Tool*. The *CANopen Design Tool* generates the following in the header file **cal_conf.h:**

```
#define CONFIG_CAN_T_CLK 20
```

This enables the bit-timing for a CAN clock of 20 MHz in the driver header file. If there is no table for the specific CAN clock a new table has to be defined. The table shall not be defined in the generic header files so that changes are not overwritten with a driver update. An adequate place for the new table is an application header file. This can be setup in the *CANopen Design Tool* which in turn generates an include statement in the header file **cal_conf.h.** All CANopen bit rates have to be defined in the bit-timing table.

To simplify the generation of a new bit-timing table there is an internet form that calculates bit-timing values for a given CAN clock. The internet address is:

http://www.port.de/engl/canprod/sv_req_form.html

The CAN clock is the clock before the prescaler.

The bit-timing table is an array for the different bit rates which is used by the function:

```
void * getBitTiming(
    UNSIGNED16 rate,  /* Baudrate (125 == 125kbit/s )*/
    void * p_usr_tab  /* NULL == internal table */
)
```

The table is located in a driver source file. In case of the TwinCAN it is located in the file **can_twincan.c**. For the TwinCAN controller with a CAN clock of 20 MHz the following values are valid. These are located in the header file **can_twincan.h**:

```
#   define CAN_BTR0_10K          0 /* Not possible */
#   define CAN_BTR1_10K          0 /* Not possible */
#   define CAN_BTR0_20K       0x31
#   define CAN_BTR1_20K       0x2f
#   define CAN_BTR0_50K       0x18
#   define CAN_BTR1_50K       0x1c
#   define CAN_BTR0_100K         9
#   define CAN_BTR1_100K      0x2f
#   define CAN_BTR0_125K         9
#   define CAN_BTR1_125K      0x1c
#   define CAN_BTR0_250K         4
#   define CAN_BTR1_250K      0x1c
#   define CAN_BTR0_500K         1
#   define CAN_BTR1_500K      0x2f
#   define CAN_BTR0_800K         0
#   define CAN_BTR1_800K      0x7f
#   define CAN_BTR0_1000K        0
#   define CAN_BTR1_1000K     0x2f
```

Listing 32, Bit-timing Definition

5.1.7. Specials about using Remote Frames (RTR)

Between CAN controllers the implementation of the RTR support differs. This is one reason that the CiA recommends to implement devices without RTR support. On the CiA web site a document is available on this subject (see Application note 802). In general a device should not answer a Remote Request in every case. The CANopen Library needs to have the complete control about the messages a device sends. With the new CANopen Library version Remote Requests are only answered by software. For CAN controllers that can only answer by hardware, RTR support is not possible. The CANopen Library uses the RTR settings from the object dictionary of the delivered device. It is possible to set the *RTR not allowed* bits to disable the RTR support.

5.2. CPU/RTOS Driver

The CPU/RTOS driver is responsible for memory management, timer functionalities, CAN-ISR management and the CAN controller access. It is coded in target specific modules like **drivers/<target-name>/cpu.c**. Functions that are valid for a complete CPU family are in **drivers/shar_src/cpu_<cpu-familie>.c**. Functions used by all drivers are placed in **drivers/shar_src/<module>.c**.

Timer Functions:

- *InitTimer*()
- *ReleaseTimer*()
- *Timer_int*()

The functions ensure the initialization/de-initialization of a hardware or software timer and the time triggered CANopen Library functionality. The CANopen function *Timer_int*() is called by an ISR or is driven by an operating system timer event. The complete time triggered functionality is shown in figure 51. It only increments the counter `coTimerTicks` and sets a global flag.

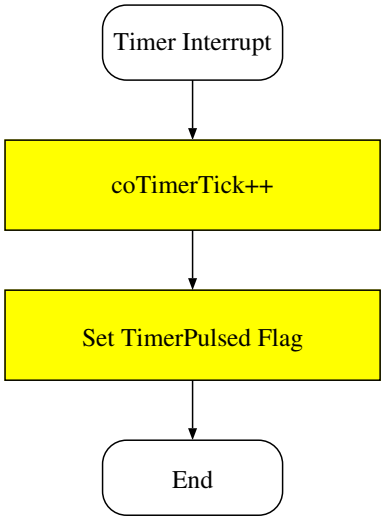


Figure 51, Time Triggered Functionality

The functionality of the *Timer_int*() can also be integrated into a user defined timer function which is called cyclically.

5.2.1. Timer XC164

For the XC164 the timer handling is implemented in **drivers/shar_src/cpu_166.c**. It provides a first initial operation. In the *CANopen Design Tool* the option *"Use pre-configured timer"* has to be enabled. This generates in the header file **cal_conf.h**:

```
#define CONFIG_COLIB_TIMER
```

The prepared timer functionality uses Timer 4. It is used as overflow timer with a timer period of 26.2 ms. This time base is used by the CANopen Library in the constant *coTimerPulse* in the unit of 1/10 ms. In the *CANopen Design Tool* this value is specified in the input field *"Timer Period"*. The value is used in the define

```
#define CONFIG_TIMER_INC 262.
```

In file **cpu_166.c** the define is assigned to the variable *coTimerPulse*.

```
UNSIGNED16 CO_CONST coTimerPulse = CONFIG_TIMER_INC;
```

If a user defined timer should be used then the define `CONFIG_COLIB_TIMER` shall not be set. The user timer interrupt service routine only needs to increment the global variable *coTimerTicks* and set a global flag with

```
SET_COLIB_FLAG(COFLAG_TIMER_PULSED);
```

to signal the CANopen Library that the Timer interrupt was triggered.

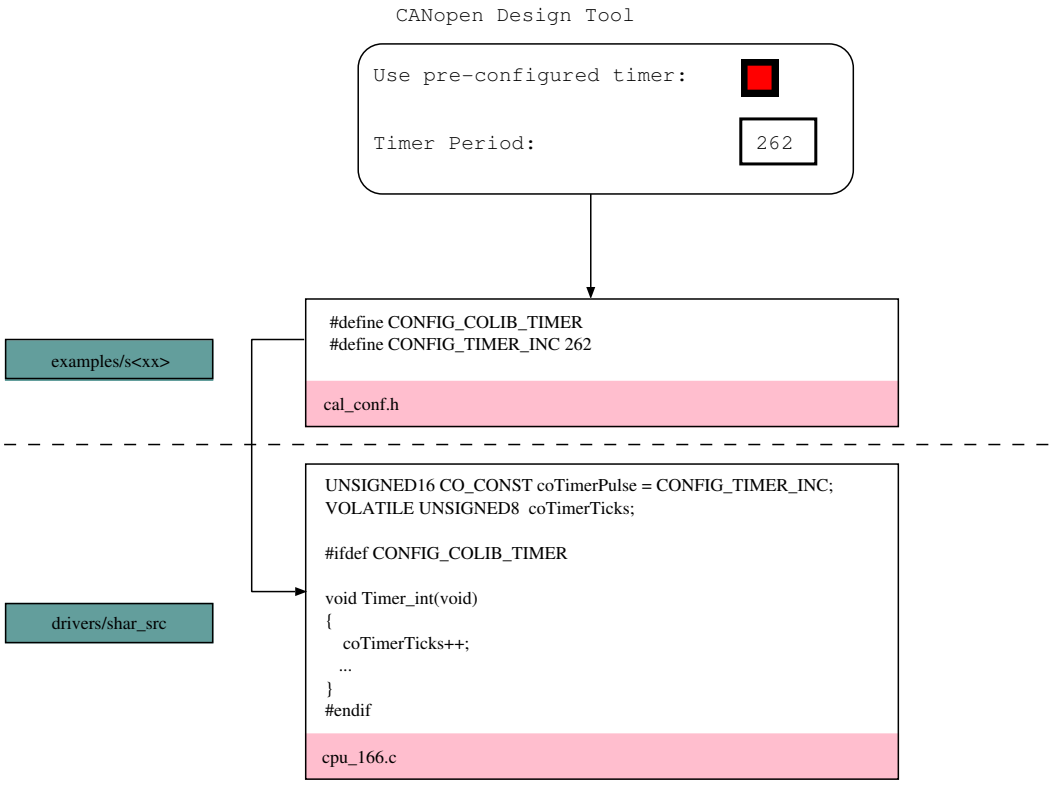


Figure 52, Timer Definitions

5.2.2. Customer Timer Implementation

Within many implementations no timer resource is free, especially the timer used in the default implementation. The CANopen Library needs only a function that is called with a constant period. It is also possible to use an already used timer interrupt. Within this periodic called function the variable *coTimerTicks* must be incremented and the CANopen Library must be informed by calling

```
coTimerTicks++;
SET_COLIB_FLAG(COFLAG_TIMER_PULSED);
```

➥ Note: Please have a look in the delivered default implementation of the timer interrupt or the code fragment within the generic driver implementation.

For using an own implementation the default implementation must be deactivated in the hardware settings. Within the *CANopen Design Tool* the setting *Use pre-configured timer* must be deactivate. This will disable the define `CONFIG_COLIB_TIMER` in the file **cal_conf.h**. The Timer period must set to the new period.

5.2.3. ISR Management

- *Init_CPU_Interrupts*()
- *Enable_CPU_Interrupts*()
- *Disable_CPU_Interrupts*()
- *Restore_CPU_Interrupts*()

These functions influence the CPU interrupt.

➥ Note: Locking of CPU interrupts and CAN interrupts can be carried out nested.

5.3. Compiler Adaptations

Compilers support different memory models. For the CANopen Library a memory model has to be chosen that allows access of a generic data pointer (unsigned char *) similarly to

- global variables (e.g. canMsg),
- object directory (possibly in Flash memory),
- object directory description structures (possibly in Flash memory),

The CAN controller of the XC164CS is outside of the addressable area of the often used memory model `LARGE`. In order not to use a bigger memory model access to the CAN controller is carried out with a far pointer. For this purpose compiler dependent definitions are used in the header file **drivers/shar_inc/co_keil.h.**

```
#define FAR  far
#define NEAR near
```

Some compiler require special constructs when a constant has to be linked into flash memory. This is not necessary for the XC164. It is sufficient to define

```
#define CO_CONST  const.
```

5.4. Application Dependent Adaptations

➥ Note: Additional hints about the default implementation of different drivers are placed in the README files within the drivers directories.

The hardware dependent adjustments to the driver are located in the directory *drivers/xc164*. This makes it possible to overwrite the actual driver due to an update in the directory *drivers/shar_inc* and *drivers/shar_src* without carrying out the adjustments again.

The file **init_xc164.c** contains the hardware initialization function *iniDevice()* and the wrapper function *initCan()*. This wrapper function is needed only by some hardware architectures. For external CAN controller the access macros have to be adapted and the hardware connection like chip selects, timings have to be initialized.
The file **cpu.c** contains the locking and releasing of CAN interrupts. It is important to note that functions that lock the interrupt with the macro *DISABLE_CAN_INTERRUPTS*() can be called nested. Therefore, releasing of CAN interrupts typically should be done with the macro *RESTORE_CAN_INTERRUPTS*(). This ensures that the CAN interrupt is released in the top most function that initially locked the interrupt.

5.5. Initial Operation

Hints for the initial operation of the prepared CAN driver. It is assumed that

- the hardware has been initialized
- the bit-timing table has been adapted,
- access macros have been selected correctly or adapted respectively.

The emphasis of this chapter lies on checking of the adjustments.

The *CANopen Design Tool* provides the possibility to enable debug settings for testing the adjustments. Debug settings are activated by the option *"Debug Settings"*. This enables the option *"Send a test message after Init"*. If this option is activated the following definitions are generated in the header file **cal_conf.h**:

```
#define CONFIG_EXPERIMENTAL 1
#define CONFIG_CAN_TX_TEST  1
```

This option causes that within the function *Init_CAN()* a message is sent with the COB-ID 100 and one data byte that has the value AA_h. It is sent with the chosen CAN bit rate. No interrupts are used.

➥ Note: After this test the options has to be deactivated again.

6. CANopen Library on Multi-Tasking Systems

This chapter describes the usage of the CANopen Library on multi-tasking systems. Multi-tasking systems include multi-tasking operating systems but also interrupt driven applications without any operating system i.e. application with timer triggered control loop.

Prepared solutions of the security mechanism for shared resources of the object dictionary are shown. Furthermore possible communication variants are discussed.

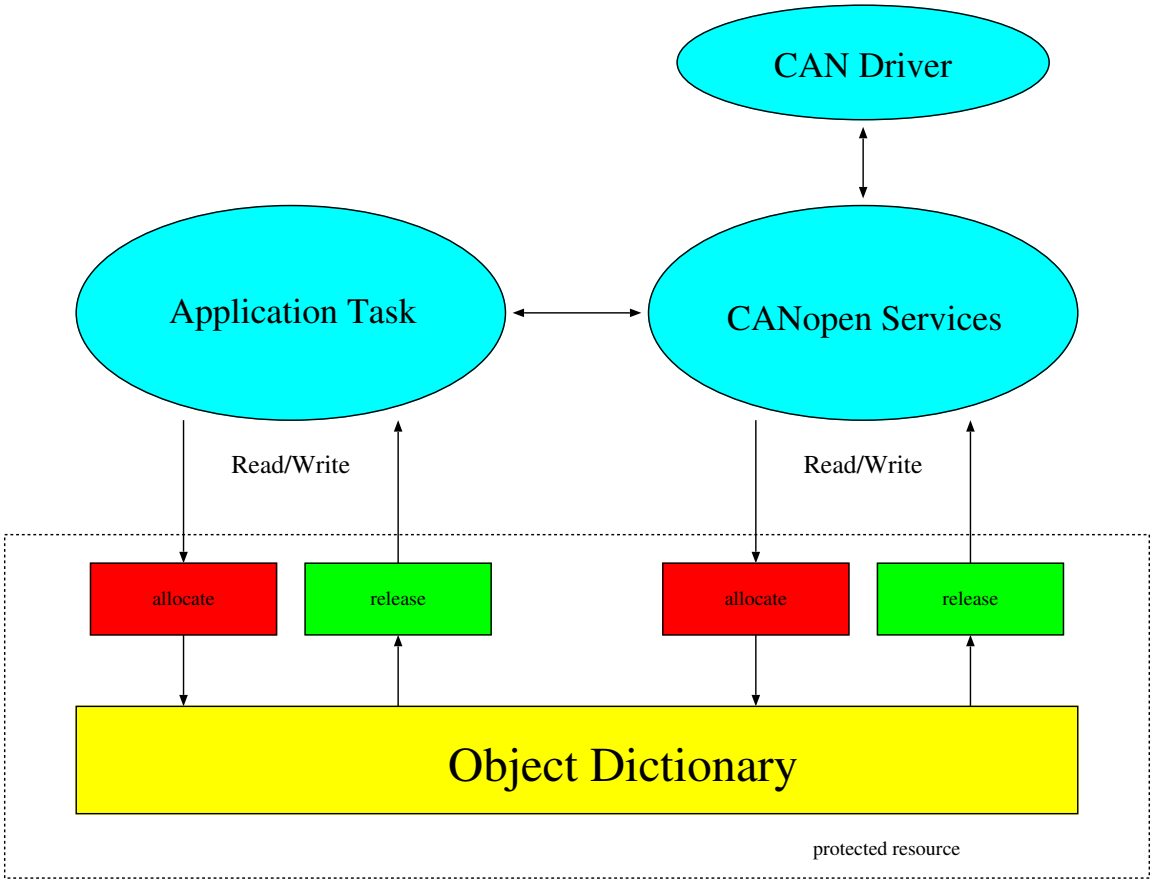


Figure 53, Security Mechanism for RTOS

A user application on a multi-tasking system consists of at least 2 processes: the application task and the CANopen communication task (figure 53). The variables of the object dictionary should be accessible by both the application and the communication task. Therefore it is necessary to protect the object dictionary. This resource has to be allocated for each read or write access. The resource is blocked until the access is finished. Blocking of the resource during read/ write access for other processes ensures that the data are always consistent.

Sometimes it is useful to have shadow variable segments e.g. if certain parameters should become valid on command. In this case, it is only to ensure that in the moment of the data updating no access is possible. The update can be done by switching between the

two data buffers (new pointer value assignment) or by copying data.

If more than one application task uses the services of the CANopen task, all CANopen services have to be protected because they are **not** reentrant. Two possibilities to use the CANopen functions are provided: firstly the direct usage (figure 53) and secondly to use a message distributor (figure 54, 55).

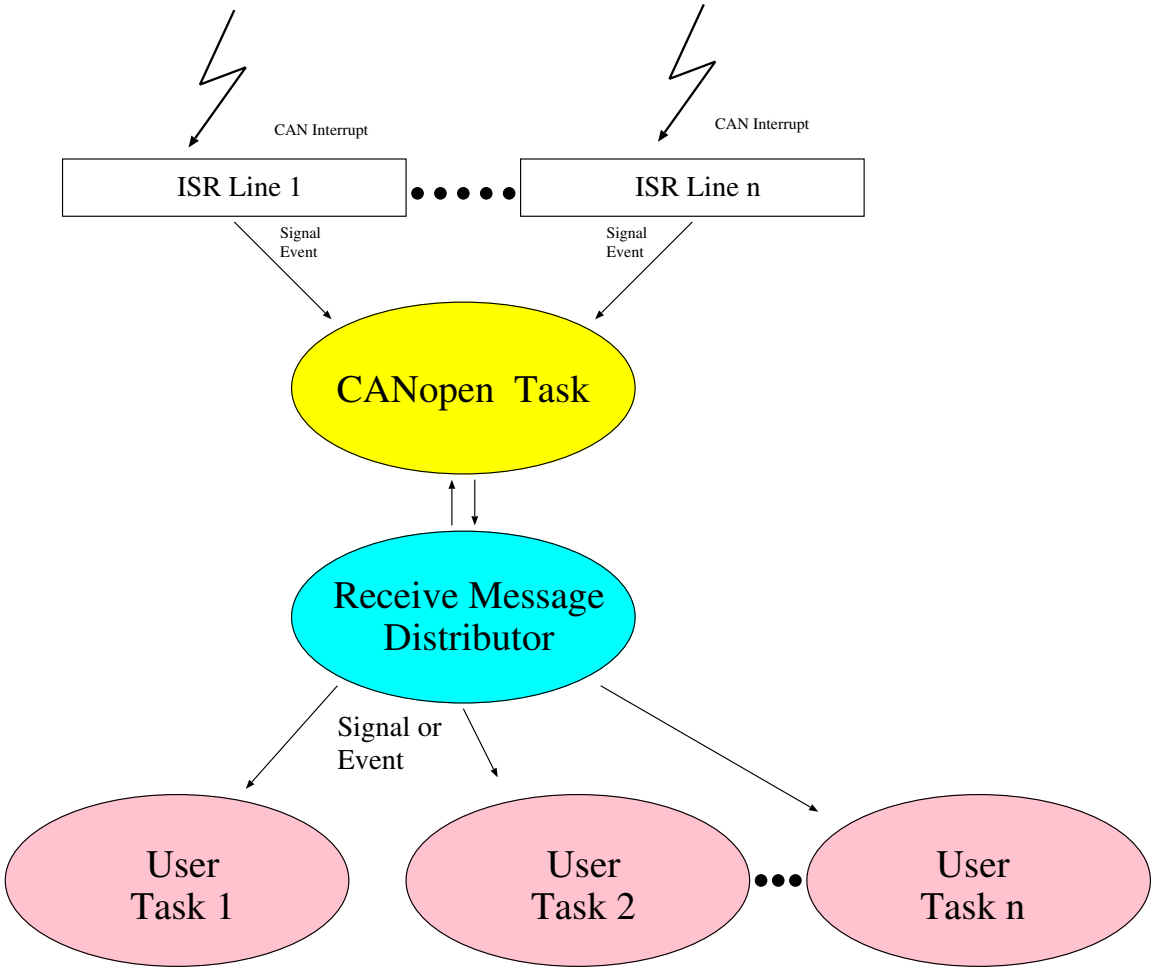


Figure 54, Message Reception on RTOS

The advantage of using a message distributor is the decoupling of application tasks from the CANopen Library. Only the message distributor programmer needs CANopen and the CANopen Library knowledge. All other application programmers can work with the commonly used mechanisms.

A further advantage is the easy switching to another communication system by replacing the distributor. So it is possible to support more then one field bus with one application software. The message distributor is a process which uses the interprocess communication mechanism of the operating system to inform the application about new messages (figure 54). For that purpose the application can read the new value from the object dictionary or the value can be sent via a queue mechanism. For data transmission the application task sends a message to the distributor. The distributor is responsible for the

message handling to and from the CANopen Library. It manages all CANopen Library function calls and can schedule the message transmission order.

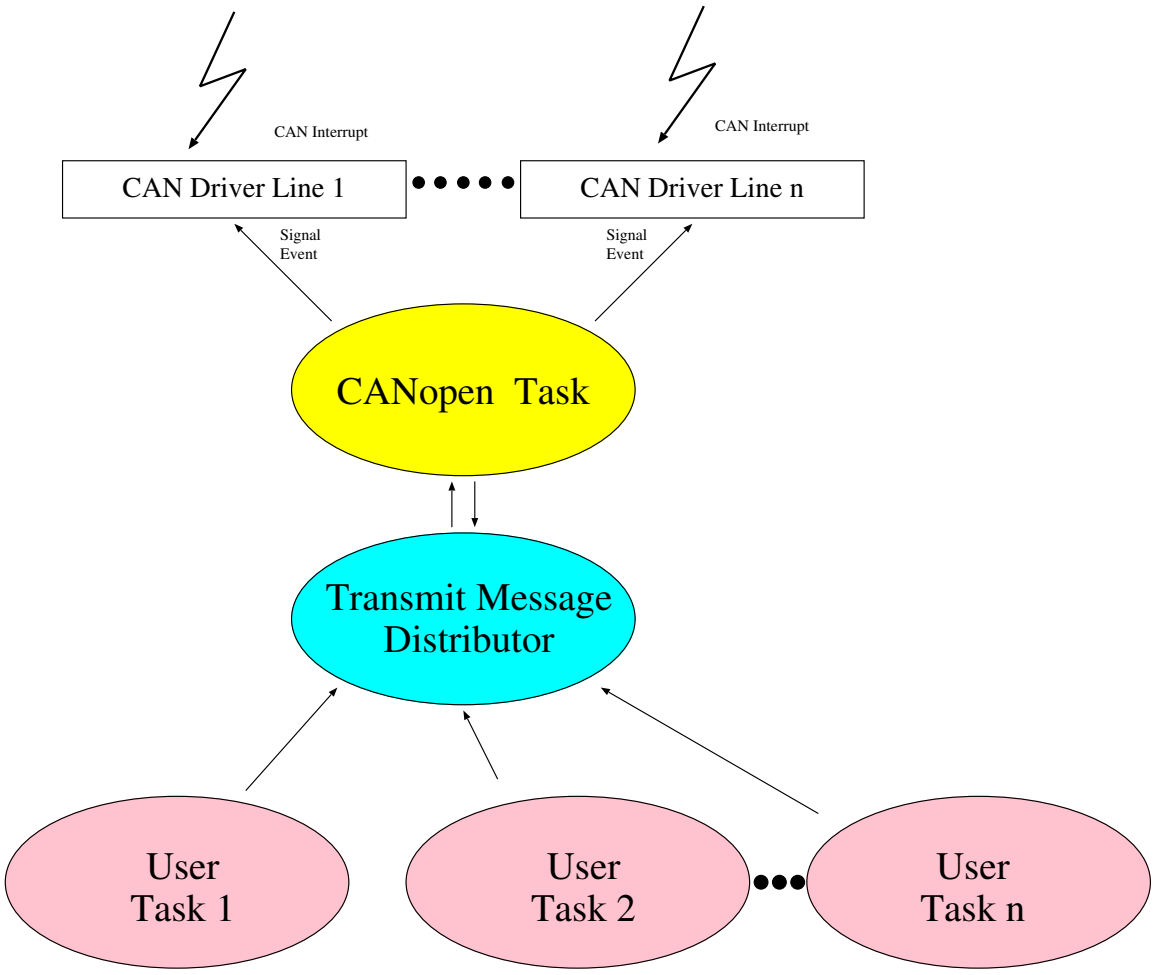


Figure 55, Message Transmission on RTOS

For resource protection many mechanisms are prepared within the CANopen Library by *port*. The following macro defines can be used to adapt the protection mechanism to your application and operating system needs.

- `CO_NEW_RX_MSG(CO_LINE)`
- `CO_COM_PART_ALLOC(CO_LINE)`
- `CO_COM_PART_RELEASE(CO_LINE)`
- `CO_APPL_PART_ALLOC(CO_LINE)`
- `CO_APPL_PART_RELEASE(CO_LINE)`

The macros listed above use a `#define` for the currently number of the used CAN line. For the single line version the `CO_LINE` define is empty. The task of the `CO_NEW_RX_MSG` macro is to inform the communication task that a new CAN message or a new error message is in the receive queue (waking up of CANopen task).
The other macros are necessary for allocating and releasing the object dictionary. The

communication and application part can be protected separately. The listing 33 shows this mechanism of the example of RTX51 for the multi line version. For the single line version the parameter `canLine` is not necessary.

```
/* use semaphore for protection */
#define CO_SEMA_L0      13
#define CO_SEMA_L1      14

#define CO_APPL_PART_ALLOC(canLine)             \
          if(canLine == 0)                      \
              os_wait(K_MBX+CO_SEMA_L0,0,NULL) \
          else                                  \
              os_wait(K_MBX+CO_SEMA_L1,0,NULL)

#define CO_APPL_PART_RELEASE(canLine)           \
          if(canLine == 0)                      \
              os_send_token(CO_SEMA_L0)         \
          else                                  \
              os_send_token(CO_SEMA_L1)

void resetActualVelocity(void)
{
    /* allocate od application part */
    CO_APPL_PART_ALLOC(0);
    /* reset velocity value (CAN line 0)*/
    l0_actual_velocity = 0;
    /* release od application part */
    CO_APPL_PART_RELEASE(0);
}
```

Listing 33, Resource Protection Example for RTX51

➥ Between allocation and release of a resource only a few instructions should be made, in order to prevent unnecessary blocking of other tasks.

```
#define CO_SEMA_L0              13
#define CO_SEMA_L1              14

#define CO_NEW_RX_MSG(canLine)          \
          if(canLine == 0)              \
              os_send_token(CO_SEMA_L0); \
          else                          \
              os_send_token(CO_SEMA_L1);

while (1)
{
    /*
     * sleep while no new message on line 0
     *
     * CANopen task should be waken up:
     *  - if a new message is received
     *  - if the CANopen timer is expired
     *    in order to check all CANopen timers
     */
    os_wait(K_MBX+CO_SEMA_L0,0,NULL);
    /*
     * interpret message for line 0
     * or handle CANopen timers
     */
    FlushMbox(0);
}
```

Listing 34, Process Activation Example for RTX51

There are no prepared mechanisms of resource protection within the functions of the CANopen Library. The user is responsible for ensuring that these functions are not interrupted.

One relatively simple way to make a CANopen Library thread-safe is to create a single mutex, lock it upon each entry to the library, and unlock it upon each exit from the CANopen Library.

For single tasking systems, which use the same resources within the application and the interrupt service routines, equivalent protection mechanism have to be used. The easiest way is to disable the interrupt(s) in the allocation macros.

7. Multi-Line Version

This chapter describes the properties of *port*'s multi-line version. With this version a single process application can access more than one CAN network (CAN line). So it is possible to implement CANopen for multiple CAN lines on target platforms without an operating system, with a single-tasking operating system or a multi-tasking operating system with a reduced resource protection mechanism. Such applications can be devices, which use an internal and an external CANopen network e.g. production cells, robots etc. Furthermore it is possible to build gateways between several networks. The CANopen network functionality (master/slave) can be different for each CAN line. Other typical applications are data loggers, monitoring systems and global network masters.

➥ An easy way to build a multi-line system is to use a PC with more than one CAN card or multi-port CAN cards.

A multi-CAN line device has <*number of CAN lines*> object dictionaries (Figure 56). This means that the device behavior can be different for each line.

There are two special defines for the multi line usage:

CONFIG_MULT_LINES	define for maximum number of CAN lines
CO_MAX_CAN_LINES	is used for actual number of CAN lines

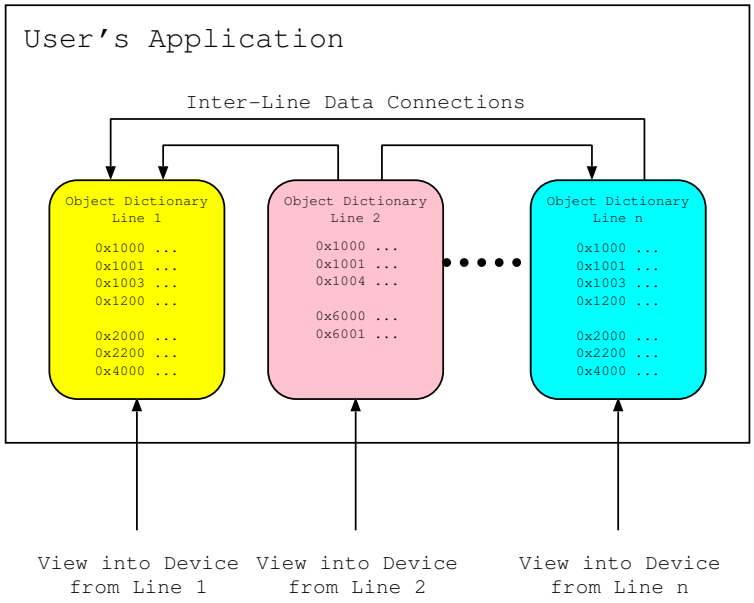


Figure 56, Multi-Line Object Dictionary

If a device variable should be accessible by two or more lines, the address references in the object dictionary of all lines have to point to the same variable (Listing 35). The object dictionary index of this variable can be different on every line.

```
/* values line 0 */
UNSIGNED32      l0_p301_device_type = 0x00000000UL;
UNSIGNED8       l0_p301_error_register = 0x00;
UNSIGNED8       myInterlineValue = 0x00;

/* values line 1 */
UNSIGNED32      l1_p301_device_type = 0x00000000UL;
UNSIGNED8       l1_p301_error_register = 0x00;

/* definition of object dictionary line 0 */

OBJDIR_T objDirLine0[] = {
     { (UNSIGNED8 *)&l0_p301_device_type,
                l0_p301_device_type_desc,
                0x1000, 1 }
    ,{ (UNSIGNED8 *)&l0_p301_error_register,
                l0_p301_error_register_desc,
                0x1001, 1 }
    , ...
    ,{ (UNSIGNED8 *)&myInterLineValue,
                l0_myInterLineValue_desc,
                0x2200, 1 }
}
/* definition of object dictionary line 1 */

OBJDIR_T objDirLine1[] = {
     { (UNSIGNED8 *)&l1_p301_device_type,
                l1_p301_device_type_desc,
                0x1000, 1 }
    ,{ (UNSIGNED8 *)&l1_301_error_register,
                l1_p301_error_register_desc,
                0x1001, 1 }
    , ...
    /* link to variable of line 0 */
    ,{ (UNSIGNED8 *)myInterLineValue,
                l1_myInterLineValue_desc,
                0x4200, 1 }
    /* end of link to variable of line 0 */

    ,{ (UNSIGNED8 *)&l1_otherVal,
                l1_otherVal_desc,
                0x5001, 1 }
}

/* object dictionary manager */

OBJDIR_T   *objDirMan[2] = { objDirLine0 , objDirLine1 };
```

Listing 35, Multi-Line Object Dictionary

The data flow within a multiple CAN line device is shown in figure 57.

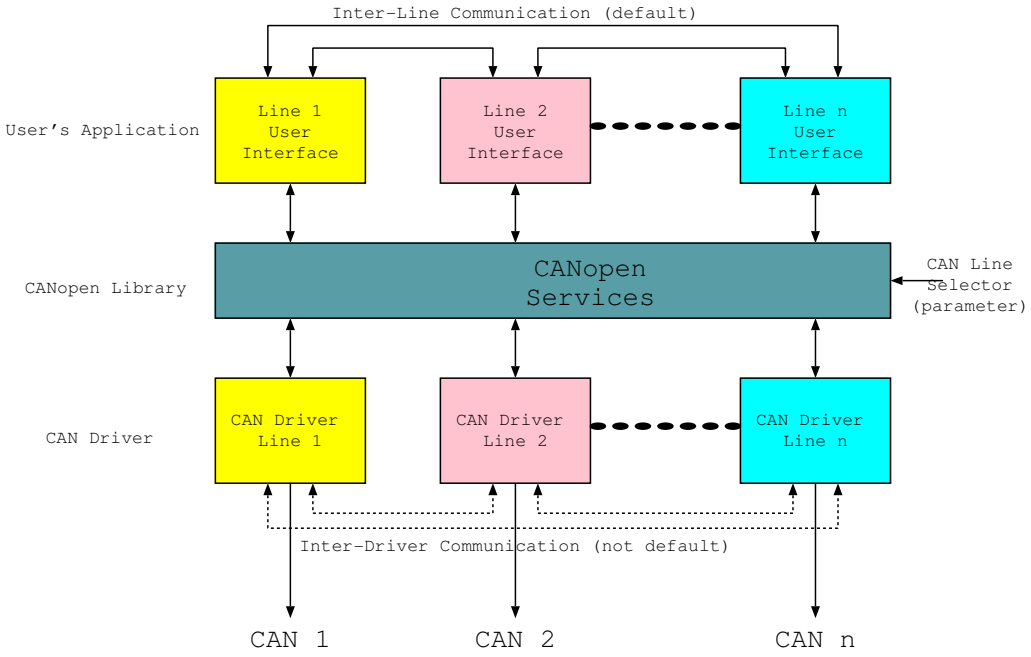


Figure 57, Multi-Line Software Layers

The figure shows the software layers and the inter-line communication of the multi-line version. Every CAN line has its own CAN driver module. For identical hardware on each line the same module can be used. In this case only the CAN controller address has to be switched and the received message put into the corresponding receive buffer.
The chosen driver concept ensures that the CAN controller can be different on every line. If all CAN controllers are equal, code size can be saved. Furthermore an inter-driver communication is possible. The reason for a communication like this is to realize a configuration gateway. Such a gateway is the basis for configuration of all nodes in all connected networks by one network participant. Usually the gateway has to have a shadow object dictionary of each node to parameterize these nodes (see inter-line communication). If any object dictionary is changed the gateway shadow dictionary has to receive an update. An update means a new configuration of the gateway (device must be programmable) or new compilation of the software (not practicable).

The condition for doing this is to have one on the side which has to be configured and one server SDO (default) on the side of the configuration software. A SDO connection has to be established via a control word which contains the line number and the node-ID. It has to be ensured that this connection is only possible in the PRE-OPERATIONAL state in order to prevent application errors in a certain CAN line. In this bypass mode received server messages are transmitted as client messages and all client messages received are transmitted as server answers to the originator. The easiest way to do this is to copy the data from the receive buffer of the one line to the transmit buffer of the other line. After the configuration, the bypass mode should be exited.

➥ All functionality which is coded into the driver is hardware dependent and mostly not portable. Such message handling is only an exception for weak performance

micro controllers.

Each message from a CAN line goes via the CAN driver and the CANopen Library to user indication CANopen Library interfaces. These interfaces are the indication functions i.e. *pdoInd*(). All of these functions have a parameter `canLine` which gives the information about the source line. For the request interfaces e.g. *writePdoReq*() the parameter `canLine` selects the destination line (CAN line selector).
It is very easy to distribute messages over various CAN networks via this mechanism. The transmission of a message to another line can be triggered with the reception of a message (Listing 36).

```
/************************************************************
*
* pdoInd - PDO indication function
*
* sends PDO 2 on line 1 if PDO 1 has been received on CAN line 0
*
* RETURNS
* .TP
* nothing
*/
void pdoInd(
     UNSIGNED16 pdoNr,  /* number of PDO */
#ifdef CONFIG_MULT_LINES
     ,UNSIGNED8 canLine  /**< number of CAN line 0..CONFIG_MULT_LINES-1 */
#endif
     )
{
    if (canLine == 0 && pdoNr == 1)
    {
              /* pdoNr., canLine */
        writePdoReq( 2, 1);
    }
}
```

Listing 36, Inter-Line Communication

8. How to Make an Application

This chapter describes the design flow for a CANopen device by using the CANopen Library provided by *port*. The way to built a NMT master and a NMT slave application will be shown by the delivered examples **s1** for the NMT slave and parts of **m1** for the NMT master. Both examples are located in the **example** directory.

The design is separated in the following steps:

- decision about the kind of device (master/slave)
- pre-definition of the CANopen services (number, properties)
- decision about the target system (hardware/operating system)

After that the user knows the device properties and can start with the coding of the communication part of his application. The necessary steps are listed below:

- preparation
- configuration of the hardware
- building up the object dictionary
- coding of the main routine
- coding of the reset behavior
- coding of the indication behavior
- optimization

8.1. Preparations

The easiest way to start programming of a new project is to use an existing example delivered by the CANopen Library. All examples are located at the directory **examples**. The *Readme* file there shows the main features for all available examples. Furthermore the directory **examples/template** contains skeletons for master and slave applications and for all indication functions. The user only has to fill these function bodies with his own needs. The names of the needed files for your application are shown in table 48.

For the first step we suggest using a slave example **s1**. This example can be compiled within its directory with the delivered make and project files.

Module	Cat.	Remark
main.c	x	Main module
usr_301.c	x	user's service indication function for CiA-301 services
usr_302.c	(x)	user's service indication function for CiA-302 services
usr_303.c	(x)	user's service indication function for CiA-303 services
usr_304.c	(x)	user's service indication function for CiA-304 services
usr_305.c	(x)	user's service indication function for CiA-305 services
nmtslave.c	x	network management behavior functions
cal_conf.h	x	contains the configuration of the CANopen Library in general this file is generated by the *CANopen Design Tool*
objects.h	x	contains the object dictionary in general this file is generated by the *CANopen Design Tool*
co_init.c	x	contains the initialization of the CANopen Library in general this file is generated by the *CANopen Design Tool*

Cat. ... category, x ... mandatory files, (x) ... optional files

Table 48, Validity of User Templates

The next step is to build a make file or project file. Within this file all dependencies of your project files should be set[9] .

Furthermore the search paths for `#include` files of your makefile or development environment are to be set to:

- *your working directory*
  to ensure that the **cal_conf.h** there is included first before all other include files
- *canopen/include*
- *drivers/shar_inc*
- *drivers/<target>*

8.2. Configuration of the Hardware

The configuration of the hardware can be done by using the *CANopen Design Tool*. All settings are saved at the header file **cal_conf.h** as compiler defines.

8.2.1. Usage of the CANopen Design Tool

The *CANopen Design Tool* simplifies the configuration of the CANopen Library and the creation of the object dictionary. It generates header files (**cal_conf.h**,**objects.h**) for the CANopen Library, an initialization file (**co_init.c**), an Electronic Data Sheet(EDS) and a

[9] The dependencies of the CANopen Library files are listed in the appendix.

documentation of the implemented objects.

The light edition of the *CANopen Design Tool* is delivered with the CANopen Library and generates only the CANopen Library configuration file **cal_conf.h**.

The basic steps to create a configuration file with the *CANopen Design Tool* Light are explained in the following paragraphs. For detailed information see the user manual of the *CANopen Design Tool* or look in the appendix (Tools).

8.2.1.1. General Settings

The menu point *General Settings* contains the main options for the CANopen node. Here you have to define the type of the node (master or slave) and other general settings. A detailed online help for each `#define` is in the *CANopen Design Tool* help menu "List of Compiler directives" available.

8.2.1.2. Hardware Settings

Here you can import an existing configuration or specify a new. In the sub-menu *Debug Settings* `#defines` for development and debugging are defined. Using it will generate additionally debugging code for the CPU and the can driver. See the source code or the Context-Help in *CANopen Design Tool* for further information and be careful if you use it.

The sub-menu *CPU settings* contains the available CPU driver modules. All CPU specific settings can be set here.

The sub-menu *Compiler Settings* contains settings to specify the used compiler. Further the alignment can be set here.

The sub-menu *CAN Settings* contains the available can driver modules. Further options that have to be set are the mode of operating for Full-CAN controller, the access type for the CAN controller, the CAN controller address and the CAN controller buffer size.

For multi-line configurations the CAN settings have to be configured for each line.

8.2.1.3. Object Dictionary Configuration

The `#defines` to enable and to configure the CANopen services are derived from the object dictionary. There the used number of CAN lines must be created and the Communication Segment of each line must be filled. The fastest way to fill the Communication Segment is to import the objects from the CiA-S301 profile database. Select the Communication Segment in the object tree, click on "Import Data from File" and select the file **profile301.pro**.

➥ There is no need to configure the remaining sections, if the *CANopen Design Tool* Light is used.

After the configuration is done save it in a project file (.can) and start the generation of the file **cal_conf.h**.

The example s1 is a simple slave device, which supports Heartbeat, 1 server SDO and 1 Receive PDO. The listing of **cal_conf.h** for it is shown below (Listing 37).

```
/*
 * CANopen Library V 4.5
 * Design Tool Light 2.3
 *
 * Automatically generated C config: don't edit
 * 10-07-2005 03:03PM
 */

#ifndef __CAL_CONF_H
 #define __CAL_CONF_H
 #define CONFIG_DESIGNTOOL_VERSION 0x0202

/* active configuration : 0 */
 #define CONFIG_USE_TARGET_0 1

/*
 * General Settings
 */
 #define CONFIG_CAN_ERROR_HANDLING              1
 #define CONFIG_FAST_SORT                       1
 #define CONFIG_SLAVE                           1
 #define CONFIG_CAN_OBJECTS                     8

/*
 * Hardware configuration 0: 0
 */

/*
 * Code Maturity Level Options
 */

/*
 * CPU Setup
 */
 #ifdef CONFIG_USE_TARGET_0
  #define CONFIG_COLIB_TIMER                    1
  #define CONFIG_CPU_FAMILY_HCS12               1
  #define CONFIG_CPU_TYPE_HCS12                 1
  #define CONFIG_TIMER_INC                      28
  # ifdef DEF_HW_PART
  #  include <cpu_hcs12.h>
 # endif /* DEF_HW_PART */

#endif /* CONFIG_USE_TARGET_0 */

/*
 * CAN Controller Setup
 */
 #ifdef CONFIG_USE_TARGET_0
  #define CONFIG_CAN_FAMILY_MSCAN               1
  #define CONFIG_COLIB_BUFFER                   1
  #define CONFIG_COLIB_FLUSHMBOX                1
  #define CONFIG_RX_BUFFER_SIZE                 10
```

```
  #define CONFIG_TX_BUFFER_SIZE                10
  #define CONFIG_CAN_START_TYPE                1
  #define CONFIG_CAN_ACCESS_MEM_MAP            1
  #define CONFIG_CAN_HCS12_NUMBER              0
  #define CONFIG_CAN_REGISTER_OFFSET           1
  #define CONFIG_CAN_TYPE_HCS12                1
  #define CONFIG_CAN_T_CLK                     8
  #define CONFIG_CAN_USE_MEMCPY                1
  #define CONFIG_STANDARD_IDENTIFIER           1
  #define CONFIG_CAN_ADDR                      {(void  *) 0x0140}
 #endif /* CONFIG_USE_TARGET_0 */

/*
 * Compiler Setup
 */
 #ifdef CONFIG_USE_TARGET_0
  #define CONFIG_ALIGNMENT                     1
  #define CONFIG_BIG_ENDIAN                    1
  #define CONFIG_COMPILER_CW_HC12              1
  #  include <co_codewarrior.h>
 #endif /* CONFIG_USE_TARGET_0 */

/*
 * CANopen Services
 */
 #define CONFIG_HEARTBEAT_PRODUCER             1
 #define CONFIG_MAPPING_CNT                    2
 #define CONFIG_PDO_CONSUMER                   1
 #define CONFIG_SDO_SERVER                     1

/*
 * Additional CANopen Settings
 */
 #define CONFIG_CONST_OBJDIR                   1

 #endif /* __CAL_CONF_H */
```

Listing 37, Part of Configuration Header for Example **s1**

For this example listing, all hardware specific defines are set for a HCS12 board.
The settings have to be adapted for the specific hardware.

8.3. Building the Object Dictionary

The object dictionary has to be filled with the necessary variables for the CANopen communication profile and for the user's application or device profile. Its structure is described in chapter 3. For each variable the user has to define:

- the index number
- the number of elements (structure or field members)

- the reference to the variable
- the reference to the variables description field

The parameters described above ensure the access to the variable[10] via the index. The access via sub-index and the test for limits, read/write permission is realized by the description field. Each entry of this field contains information about one sub-index entry. This information is necessary to build a robust interface for the device.

➥ The *CANopen Design Tool* is available for an automatic generation of an object dictionary, in the file **objects.h**.

The example **s1** uses only 2 device profile variables.

Index	Subindex	Name	Data Type	Assignment
0x6200	1	*p401_write_state_8*	UNSIGNED8	RPDO 1
0x6202	2	*p401_polarity_write_8*	UNSIGNED8	

Table 49, Device Profile Variables of Example s1

The listing 38 shows the implementation of the user variables and the PDO parameters. The object dictionary implementation contains a definition part (the `#define DEF_OBJ_DIC` has to be set before **objects.h** is included) and a declaration part. In the definition part all variables are defined. The object dictionary consists of three kinds of data. The first are the user application and the second kind are CANopen communication variables. For each of these variables there is a description (<*variable name*>*_desc*). The third kind is the object dictionary **objDir.** This contains references to all variables and their descriptions.

The PDO mappings for the first TPDO and the first RPDO are set to the variables *p301_n1_transmit_pdo_mapping* and *p301_n1_receive_pdo_mapping*. Therefore it is very easy to see the assignment of application variables to the PDO. In general the application knows nothing about the communication variables.

```
/*
* objects.h - generated object dictionary for a CANopen device
*
*------------------------------------------------------------------------------
*/

/**
* \file objects.h
* \author port GmbH, Halle (Saale)
*  $Revision: 1.14 $
*  $Date: 2012/05/11 10:18:44 $
*
* This file contains the selected objects for a CANopen device.
* It was generated by the CANopen Design Tool V2.3
* by port GmbH, Halle.
* Generation: 01-19-2006 12:29
*
```

[10] A variable can be a common variable, a structure or an array.

```
*/


/*  additional typedefs to those in headerfiles*/
#ifdef DEF_OBJ_DIC
/*  number of entries in the object dictionary */
UNSIGNED16      maxObjDicElements = 10;

/*  Definition of Variables (Device Objects)  */
UNSIGNED32      p301_device_type = { 0x00020191UL };
UNSIGNED8       p301_error_register = { 0x00 };
VIS_STRING_T    p301_manu_device_name[35] =
                { "demo - port GmbH Linux Starter Kit" };
UNSIGNED16      p301_prod_hb_time = { 0x000003E8 };
IDENTITY_T      p301_identity = { 0x4, 0x34UL, 0x0UL, 0x0UL, 0x0UL };
SDO_COMM_PAR_T  p301_n1_ssdo_par = { 2, 0x00000600UL, 0x00000580UL, 0x00 };
PDO_COMM_PAR3_T p301_n1_rpdo_para = { 0x03, 0x00000200UL, 0xFE, 0x0 };
PDO_MAPPING2_T  p301_n1_rpdo_map = { 0x02, { 0x62000108UL, 0x62000208UL }};
UNSIGNED8       p401_write_state_8[3] = { 0x2, 0x00, 0x00};
UNSIGNED8       p401_polarity_write_8[3] = { 0x2, 0x00, 0x00};

/* Default values */
UNSIGNED8       l0_default0_0005[12] = { 0,4,2,3,254,2,2,0,0,2,0,0 };
UNSIGNED16      l0_default0_0006[2] = { 0x3e8,0x0 };
UNSIGNED32      l0_default0_0007[10] = { 0x20191,0x34,0x0,0x0,0x0,
                    0x600,0x580,0x200,0x62000108,0x62000208 };
STRING_DATA_T   l0_string_data[1] = {{0x23, 0x23}};

/*  Definition of Value Descriptions  */
VALUE_DESC_T    p301_device_type_desc[1] =  {
        { (UNSIGNED8 *) &l0_default0_0007[0],
        CO_TYPEDESC_UNSIGNED32, CO_WRITE_PERM | CO_READ_PERM | CO_NUM_VAL }
    };
VALUE_DESC_T    p301_error_register_desc[1] =  {
        { (UNSIGNED8 *) &l0_default0_0005[0],
        CO_TYPEDESC_UNSIGNED8, CO_READ_PERM | CO_NUM_VAL }
    };
VALUE_DESC_T    p301_manu_device_name_desc[1] =  {
        { (UNSIGNED8 *) &l0_string_data[0],
        CO_TYPEDESC_VISSTRING, CO_WRITE_PERM | CO_READ_PERM }
    };
VALUE_DESC_T    p301_prod_hb_time_desc[1] =  {
        { (UNSIGNED8 *) &l0_default0_0006[0],
        CO_TYPEDESC_UNSIGNED16,
        CO_READ_PERM | CO_WRITE_PERM | CO_NUM_VAL }
    };
VALUE_DESC_T    p301_identity_desc[5] = {
        { (UNSIGNED8 *)&l0_default0_0005[1],
        CO_TYPEDESC_UNSIGNED8, CO_READ_PERM | CO_NUM_VAL },
        { (UNSIGNED8 *)&l0_default0_0007[1],
        CO_TYPEDESC_UNSIGNED32, CO_READ_PERM | CO_NUM_VAL },
        { (UNSIGNED8 *)&l0_default0_0007[2],
        CO_TYPEDESC_UNSIGNED32, CO_READ_PERM | CO_NUM_VAL },
```

```
        { (UNSIGNED8 *)&l0_default0_0007[3],
        CO_TYPEDESC_UNSIGNED32, CO_READ_PERM | CO_NUM_VAL },
        { (UNSIGNED8 *)&l0_default0_0007[4],
        CO_TYPEDESC_UNSIGNED32, CO_READ_PERM | CO_NUM_VAL }
    };
VALUE_DESC_T    p301_n1_ssdo_par_desc[3] = {
        { (UNSIGNED8 *)&l0_default0_0005[2],
        CO_TYPEDESC_UNSIGNED8, CO_READ_PERM | CO_NUM_VAL },
        { (UNSIGNED8 *)&l0_default0_0007[5],
        CO_TYPEDESC_UNSIGNED32, CO_READ_PERM | CO_NUM_VAL },
        { (UNSIGNED8 *)&l0_default0_0007[6],
        CO_TYPEDESC_UNSIGNED32, CO_READ_PERM | CO_NUM_VAL }
    };
VALUE_DESC_T    p301_n1_rpdo_para_desc[4] = {
        { (UNSIGNED8 *)&l0_default0_0005[3],
        CO_TYPEDESC_UNSIGNED8, CO_READ_PERM | CO_NUM_VAL },
        { (UNSIGNED8 *)&l0_default0_0007[7],
        CO_TYPEDESC_UNSIGNED32,
        CO_READ_PERM | CO_WRITE_PERM | CO_NUM_VAL },
        { (UNSIGNED8 *)&l0_default0_0005[4],
        CO_TYPEDESC_UNSIGNED8,
        CO_READ_PERM | CO_WRITE_PERM | CO_NUM_VAL },
        { (UNSIGNED8 *)&l0_default0_0006[1],
        CO_TYPEDESC_UNSIGNED16,
        CO_READ_PERM | CO_WRITE_PERM | CO_NUM_VAL }
    };
VALUE_DESC_T    p301_n1_rpdo_map_desc[3] = {
        { (UNSIGNED8 *)&l0_default0_0005[5],
        CO_TYPEDESC_UNSIGNED8, CO_READ_PERM | CO_NUM_VAL },
        { (UNSIGNED8 *)&l0_default0_0007[8],
        CO_TYPEDESC_UNSIGNED32, CO_READ_PERM | CO_NUM_VAL },
        { (UNSIGNED8 *)&l0_default0_0007[9],
        CO_TYPEDESC_UNSIGNED32, CO_READ_PERM | CO_NUM_VAL }
    };
VALUE_DESC_T    p401_write_state_8_desc[3] = {
        { (UNSIGNED8 *)&l0_default0_0005[6],
        CO_TYPEDESC_UNSIGNED8, CO_READ_PERM | CO_NUM_VAL },
        { (UNSIGNED8 *)&l0_default0_0005[7],
        CO_TYPEDESC_UNSIGNED8,
        CO_MAP_PERM | CO_READ_PERM | CO_WRITE_PERM | CO_NUM_VAL },
        { (UNSIGNED8 *)&l0_default0_0005[8],
        CO_TYPEDESC_UNSIGNED8,
        CO_MAP_PERM | CO_READ_PERM | CO_WRITE_PERM | CO_NUM_VAL }
    };
VALUE_DESC_T    p401_polarity_write_8_desc[3] = {
        { (UNSIGNED8 *)&l0_default0_0005[9],
        CO_TYPEDESC_UNSIGNED8, CO_READ_PERM | CO_NUM_VAL },
        { (UNSIGNED8 *)&l0_default0_0005[10],
        CO_TYPEDESC_UNSIGNED8,
        CO_READ_PERM | CO_WRITE_PERM | CO_NUM_VAL },
        { (UNSIGNED8 *)&l0_default0_0005[11],
        CO_TYPEDESC_UNSIGNED8,
        CO_READ_PERM | CO_WRITE_PERM | CO_NUM_VAL }
```

```
    };
/* Definition of the object directory */

OBJDIR_T objDir[] = {
{   (UNSIGNED8 *)&p301_device_type,
    (VALUE_DESC_T *)p301_device_type_desc,
    0x1000, 1  }
```

Listing 38,
Parts of the Object Dictionary Implementation for Example s1

8.4. Coding of the Main Routine

There is a template (**template/main.c**) for coding the main routine. It is recommended to use this template for rapid programming. The following steps are necessary for the design of an application:

- Design of the boot-up behavior (initialization)
- Design of the application
- Design of the shutdown behavior

All differences between the master and slave applications are shown in bracket comments.

The boot-up behavior of an application required the following:

- hardware initialization (optional i.e. setting of chip selects) *iniDevice()*
- initialization of CAN controller, timer and ISR's *initCan()*
- initialization of the CANopen Library *init_CANopen*() - include reset of the object dictionary (*)
- modify communication parameter at the object dictionary if the predefined connection set should not be used (p.e. by defining INIT_USER_SETTINGS) (*)
- definition of the CANopen communication objects which should be used *defineEmcy*() (*), *defineSdo*() (*), *definePdo*() (*)
- definition of a local node *createNodeReq*() (*)
- creation of a node management list *createNetworkReq*() (only master) (*)
- insertion of information about all nodes in the network to the management list *addRemoteNodeReq*() (only master) (*)
- starting of the local node *startRemoteNodeReq*() (only master) (*)
- activation of interrupts for CAN and timer
- starting of all nodes *startRemoteNodeReq*() (only master)

All steps that are marked with **(*)** are carried out by the *CANopen Design Tool*[11]. Therefore it creates the file **co_init.c**, which provides the function *init_Library()*. In this case *init_Library()* must be called after *initCan()* .

[11] Only the standard edition, not *CANopen Design Tool* Light

The difference between a master and a slave is that the master controls all network participants. Therefore each master application has to manage the network. In the CANopen Library the master collects all communication relevant information in a node management list (network). With this list the master is able to guard the nodes and to influence their communication states. Before all network participants go to the OPERATIONAL state the master or another configuration application can parameterize their communication variables i.e. COB-ID, PDO mapping.

```
int main(void)
{
RET_T         commonRet;      /* return value for CANopen functions */
UNSIGNED8     ret;            /* return value for common purpose */
BOOL_T        err = CO_FALSE; /* error flag */

    /* Hardware Initializion; e.g SIO, Chip-Selects, ... */
    ret = iniDevice();
    PRINTRET("iniDevice: %02x\n", (int)ret);

    /*
     * CAN_START_BIT_RATE = 0       : read bit timing from init file
     * CAN_START_BIT_RATE = bitRate : use local bitRate variable
     */
    ret = initCan(CAN_START_BIT_RATE);
    PRINTRET("initCan: %02x\n", (int)ret);

    /* defines also the Network control Object -- NMT
     * reset communication and goes to the
     * state preoperational + Initialization of CANopen
     */
    commonRet = init_Library();
    PRINTRET("init_Library: 0x%02x\n", (int)commonRet);

    /*
     * timer is needed for inhibit time and
     * host life guarding (if life guarding is supported)
     */
    initTimer();

    SetIntMask();
    Start_CAN();
    ENABLE_CPU_INTERRUPTS();
}


/* Inititialization of CANopen
 * defines also the Network control Object -- NMT
 * resets communication
 */
RET_T init_Library(void)
{
    commonRet = initCANopen();
    PRINTRET("initCANopen: %02x\n", (int)commonRet);
```

```
    /* modify communication parameter,
     * if the predefined connection set shouldn't be used
     * it's not used in s1
     */
    INIT_USER_SETTINGS();

    /* Definition of CANopen objects */
    /* ============================= */

    /* definition of the 1st SDO
     * first parameter is SDO number, for later references
     * second parameter is SDO type: CLIENT | SERVER
     */
    commonRet = defineSdo(1, SERVER);
    PRINTRET("Define 1. Server SDO: %02x\n", (int)commonRet);

    /* Definition of the 1st RPDO
     * 1st parameter -  type of PDO: RECEIVE | TRANSMIT
     * 2nd parameter -  PDO number for later references
     * 3rd parameter -  permission for dynamically mapping
     */
    commonRet = definePdo(RECEIVE_PDO,1 , CO_FALSE);
    PRINTRET("Define 1. Receive PDO: %02x\n", (int)commonRet);

    /* creating a network node */
    /* definition of the local node */
                          /* Node Guarding, Heartbeat */
    commonRet = createNodeReq(CO_FALSE, CO_TRUE);
    PRINTRET("NMT Node created: %02x\n", (int)commonRet);
}
```

Listing 39, Example s1, Initialization

During the function *initCANopen()* all entries of the object dictionary are set to their default values, and all COB-IDs are set according to the predefined connection set, independ the saved entries at **objects.h**. If the application needs other values it can be set by defining *INIT_USER_SETTINGS()* as macro or as function.

The design of the application is the user's task. He has to ensure that the CAN message buffer is read continuously or event driven. The easiest way to perform this is an endless loop from which the buffer read function and the application functions are called cyclically. (listing 40).

```
while(1)
{
    /* read and interpret CAN message */
    FlushMbox();

    /* application function */
    application();
}
```

Listing 40, Endless Loop Reception Process But it is also possible to call this function by a Real Time Operating System signal or other mechanisms (listing 41). Then the user's application runs within other tasks.

```
while(1)
{
    /*
     * sleep while no CAN event
     *
     * CANopen task should be waken up:
     *  - if a new message is received
     *  - if the CANopen timer is expired
     *    in order to check all CANopen timers
     */
    os_wait(event);
    /*
     * interpret received message
     * or handle CANopen timers
     */
    FlushMbox();
}
```

Listing 41, Signal Driven Reception Process

The implementation of the shutdown behavior is only necessary if the CANopen Library can be left during the run time of the application i.e. if running on a operating system. The called functions will free the allocated system resources. The following steps are to be made for this:

- stop all nodes *stopRemoteNodeReq*() (only master)
- deactivate the ISRs for CAN and timer
- de-initialize CAN controller, timer and their ISR's
- delete the node entries *removeRemoteNodeReq*() (only master)
- delete the local node *deleteNodeReq*()
- delete the node management list (network) *deleteNetworkReq*() (only master)
- de-initialize the CANopen Library *leaveCANopen*()

```
main()
{
    ....
    /*
     * Application really ends
     *
     * release all resources.
     */
    releaseTimer();
    ResetIntMask();
    deleteNodeReq();
    /* leaves CANopen    */
    leaveCANopen();
```

```
        putchar('X');
        return(0);
    }
```

Listing 42, Example s1, Shutdown

The file **co_init.c**, generated by the *CANopen Design Tool* contains the function `deinit_Library()` that does all necessary steps to shutdown the CANopen Library.

8.5. Coding of the Reset Behavior

The reset behavior is a property only for slaves. Each slave can receive a *Reset Application* or a *Reset Communication* command from the master. In the module **nmtslave.c** the user will find the functions *resetApplInd*() and *resetCommInd*(). Within these functions an individual reset of the application data and states can be implemented. In the function for the communication reset the CAN bit rate[12] or node-ID can be changed.

Furthermore there is the function *newStateInd*(). This function informs the user's application about each transition of the communication state machine. This information can be important, because a few communication services are not available in certain states. For example, the application transmits error codes via PDOs and the master forces the node to PRE-OPERATIONAL. Then PDOs are not allowed.

8.6. Coding of the Indication Behavior

For each received CANopen message an indication function is called. Templates for these functions can be found in the module **template/usr_301.c.** The task of these functions is to start an error handling or additional post-processing or pre-processing for object dictionary entries or to initiate a reaction. There are three kinds of errors which can be handled with the indication functions. The first kind are errors in the CAN controller and CAN driver message queues. The function *canErrorInd*() informs the application about error of the CAN controller e.g. "Bus-Off", "Error passive" and about the overflow of the receive and transmit buffer since the last call to *canErrorInd()*. The current state of the CAN controller can be queried by *getCanDriverState*().

If an emergency message is received, the function *emcyInd*() is called at the EMCY consumer. There the user can handle fatal errors of remote devices.

The second kind of errors are the Node Guarding errors. In the case of a lost connection between master and slave the function *mGuardErrorInd*() is called at the master and the function *sGuardErrorInd*() at the slave.

The other functions of the module are for the common data service handling. For each received PDO the function *pdoInd*() is called. Within this function the user can define reactions which should be initiated by certain PDOs.

```
/*****************************************************************
*
```

[12] Please ensure that all devices in the network use the same CAN bit rate.

```
* pdoInd - indicates the occurrence of a PDO
*
* In this function the user has to define his application specific
* handling for PDOs.
*
* \returns nothing
*/

void pdoInd(
     UNSIGNED16 pdoNr    /* nr of PDO */
#ifdef CONFIG_MULT_LINES
     ,UNSIGNED8 canLine  /**< number of CAN line
                              0..CONFIG_MULT_LINES-1 */
#endif
    )
{
    switch(pdoNr) {
        case 1:
            /* we have a fixed PDO mapping.
             * Therefore we know that the two sub-indices of 0x6200
             * are mapped. The content now in the object directory
             * has to be transferred to the two hardware ports 1 and 2.
             */
            set_outputs(1);
            set_outputs(2);
            break;
    }
}
```

Listing 43, Example PDO Indication

The server SDO services are divided into read and write access functions. For every kind there is a function *sdoWrInd*() for write access and *sdoRdInd*() for read access. It is possible to initiate reactions or to manipulate values i.e. unit transformations with these functions.

```
/************************************************************************
* sdoWrInd - indicates the occurrence of a SDO write access
*
* \retval CO_OK success
* \retval CO_E_xxx error
*
*/

RET_T sdoWrInd(
          UNSIGNED16 index,      /* index to object */
          UNSIGNED8  subIndex   /* sub-index to object */
#ifdef CONFIG_MULT_LINES
          ,UNSIGNED8 canLine    /**< number of CAN line
                                     0..CONFIG_MULT_LINES-1 */
#endif
        )
{
    if (index == 0x6200) {
        /* write request to the digital output 8-bit ports */
        set_outputs(subIndex);
    }

    return CO_OK;

}
```

Listing 44, Example **s1,** SDO Write Indication

```
/************************************************************************
* sdoRdInd - indicates the occurrence of a SDO read access
*
* \retval CO_OK success
* \retval CO_E_xxx error
*
*
*/

RET_T sdoRdInd(
          UNSIGNED16 index,      /* index to object */
          UNSIGNED8  subIndex    /* index to object */
#ifdef CONFIG_MULT_LINES
          ,UNSIGNED8 canLine    /**< number of CAN line
                                     0..CONFIG_MULT_LINES-1 */
#endif
          )
{
    return CO_OK;
}
```

Listing 45, Example **s1,** SDO Read Indication

Further service indications and confirmations are defined within **usr_301.c**.

The function *getNodeId*() is very important. This function returns the node-ID which was read i.e. by a DIP switch or from a nonvolatile memory. It is called implicitly during the CANopen Library initialization *initCANopen*() and before executing *Reset Communication*.
The function *timeInd*() indicates a received Time Stamp.
testSdoValue() is a filter function for values which should be written to the object dictionary by SDO. Using this filter the application can check values before they are written into the object dictionary.
In a SDO client the functions *sdoWrCon*() and *sdoRdCon*() are called if the confirmation message to a SDO read or write request was received. Within these functions the application can react to the successful service or to the errors resulting from an abort domain transfer.
For a convenient usage of nonvolatile memory to store data via object 1010_h the functions *saveParameterInd*(), *clearParameterInd*() and *loadParameterInd*() have been introduced.

➥ For detailed information about the function mentioned above please have a look to the Reference Manual.

8.7. Optimization

On completion of the programming of the application behavior, all communication needs are fixed. Therefore the communication part can be optimized.
The optimization can be done in:

- **cal_conf.h**
- **objects.h**
- driver modules

In the configuration header **cal_conf.h** all services which are not used by the application can be disabled. That is a typical code size optimization.
In the object dictionary implementation, all unused variables of the communication and the application should be removed. Data memory space can be saved.
In the driver modules for CAN and the timer, all unnecessary functionalities can be removed in order to reduce code size. Furthermore the run time behavior can be optimized, for example by removing the transmit queue for Full-CAN controller.

9. Trouble Shooting

This chapter describes possible error situations. If you have trouble with the CANopen Library, please read the following error descriptions first.
If you do not find a solution for your problem in this chapter, you can request support, see chapter 1.6, or see the other appendixes for more description of internal behavior.

Please check the following points:

- Have you checked the return values of the functions?
- Are the `#defines` in **cal_conf.h** set correctly for the services used?
- Have you checked the order of function calls?
- Have you included the associated **objects.h** and **cal_conf.h**?
- Have you recompiled the sources after changing **objects.h** and **cal_conf.h**?
- Do you have an overview which functionalities should be made on your local device and which on the remote device(s)?

Detailed error situation descriptions are listed in table 50.

Description	Error Reason
no transmission of PDOs is possible	node is not in state OPERATIONAL PDO disabled, see PDO parameter PDO is only a RTR PDO `#define CONFIG_PDO_PRODUCER` is not set
no reception of PDOs is possible	node is not in state OPERATIONAL PDO disabled, see PDO parameter `#define CONFIG_PDO_CONSUMER` is not set
value of TPDO contents of remote device is wrong	dynamic mapping was not carried out your compiler does not support byte alignment and `#define CONFIG_ALIGNMENT` is not set
variable PDO mapping is not possible on remote device (Abort Domain Transfer)	mapping flag not set for selected variable (**objects.h**) at RPDO: selected variable is read only (**objects.h**) at TPDO: selected variable is write only (**objects.h**) `#define CONFIG_DYN_PDO_MAPPING` was not set `#define CONFIG_MAX_DYN_MAP_ENTRIES` was not set or their value is too low
variables contains wrong values	your compiler does not support byte alignment and `#define CONFIG_ALIGNMENT` is not set or has wrong value you have included the wrong object dictionary object dictionary variables are parameterized incorrectly

Description	Error Reason
definePdo() returns `E_MAP`	mapping entry in **objects.h** does not exist your compiler does not support byte alignment and `#define CONFIG_ALIGNMENT` is not set or has wrong value your processor uses big-endian format and `#define CONFIG_BIG_ENDIAN` is not set
device does not answer to SDO Transfer	`#define CONFIG_SDO_SERVER` is not set
device does not initiate to SDO Transfer	`#define CONFIG_SDO_CLIENT` is not set
no data transfer is possible after system integration	COB-ID of sender and receiver are not equal

Table 50, Suggestions for Trouble Shooting

10. Appendices

10.1. Appendix — Header Files

This appendix contains an overview of all header files for the CANopen Library. The location mentioned in the tables is the location of the headers within the installation path. There are 4 kinds of header files for the CANopen Library compilation.

1. project specific headers
2. shared driver headers
3. example driver headers
4. CANopen Library interface headers

Location: *<project include>*	
Name	**Description**
cal_conf.h	configuration header for CANopen Library compilation
objects.h	object dictionary implementation

Table 51, Project Specific Headers

Location: *drivers/shar_inc*	
Name	**Description**
cdriver.h	constants for all CAN controllers
cpu_xxx.h	constants for CPU driver part
can_xxx.h	constants for CAN driver part
co_xxx.h	constants for special compilers

Table 52, Shared Driver Headers

Location: *drivers/<hardware>*	
Name	**Description**
examples.h	adaptations to use our examples

Table 53, Example Driver Headers

Location: *canopen/include*	
Name	**Description**
co_acces.h	access to the object dictionary

Location: *canopen/include*	
Name	**Description**
co_cobid.h	defines of predefine COB-IDs in CiA-301
co_debug.h	defines for debug interface
co_def.h	global defines for the CANopen Library
co_drv.h	defines for driver interface
co_drvif.h	typedefs for driver interface
co_drvmc.h	additionally defines for multi can devices
co_drvry.h	additionally defines for redundancy support
co_emcy.h	defines for EMCY
co_flag.h	defines for internal flag usage
co_flyma.h	defines for Flying Master
co_guard.h	defines for Node Guarding
co_hb.h	defines for Heartbeat
co_led.h	defines for Led
co_lme.h	defines for layer management
co_lss.h	defines for LSS
co_mcpy.h	macros for memcpy
co_mpdo.h	defines for MPDOs
co_nmt.h	defines for NMT
co_nmt_m.h	defines for NMT master services
co_odidx.h	object dictionary indexes
co_pdo.h	defines for PDO
co_redcy.h	defines for redundancy support
co_sdo.h	defines for SDO
co_sdomg.h	defines for SDO Manager
co_sdorq.h	defines for SDO Requester
co_setcp.h	defines for internal communication parameter
co_splus.h	defines for SLAVE with NMT capabilities
co_srdo.h	defines for Safety Relevant Objects
co_stor.h	defines for parameter storage
co_stru.h	structure definition
co_sync.h	defines for SYNC
co_time.h	defines for time

Location: *canopen/include*	
Name	**Description**
co_timer.h	defines for timer routines
co_type.h	type definition
co_usr.h	defines for user interface
co_util.h	defines for utilities
co_vers.h	version information

Table 54, CANopen Library Interface Headers

10.2. Appendix — Data Types

This appendix describes the basic data types of the CANopen Library. All atomic data types are defined within the header file **co_type.h**. In general this file is part of the CANopen Library interface headers, but some development environments define these types, too. Therefore it is necessary to overload this file by a **co_type.h** in your project include directory.

➥ The compiler's search path has to contain your project include directory path before the CANopen Library include path.

Then your **co_type.h** file (or any other name used) is the project specific data type header file.

The following table describes the types. Using only the type definition `BOOL_T`, `UNSIGNED<x>` and `INTEGER<x>` in own applications is recommended.

Type	**Description**
`BOOL_T`	boolean type
`UNSIGNED8`	unsigned 8 bit value
`UNSIGNED16`	unsigned 16 bit value
`UNSIGNED32`	unsigned 32 bit value
`UNSIGNED64`	unsigned 64 bit value
`INTEGER8`	signed 8 bit value
`INTEGER16`	signed 16 bit value
`INTEGER32`	signed 32 bit value
`LOOPCNT_U8`	unsigned 8 bit value - register variable (only Keil C51)
`LOOPCNT_U16`	unsigned 16 bit value - register variable (only Keil C51)
`REAL32`	float 32 bit value
`VIS_STRING_T`	unsigned char
`OCT_STRING_T`	unsigned char

Type	Description
BIT_STRING_T	unsigned char
DOMAIN_T	void *
COB_KIND_T	enumeration of types of CAN messages
BASIC_DATA_T	enumeration of CANopen basic types
NODE_STATE_T	enumeration of NMT states
USER_T	enumeration for SDO types
CO_USER_T	enumeration for PDO types
RET_T	return values of CANopen functions
ERROR_SPEC_T	defines for error conditions
STATE_T	enumeration

Table 55, Atomic Types

10.3. Appendix — SDO Abort Codes

The SDO Abort Transfer is a negative conformation of a SDO request. This service contains a code, which specifies the kind of the abort. The CANopen Library by *port* supports the following SDO abort codes:

SDO Abort Code	RET_T Value	Description
0504 0000h	CO_E_SDO_TIMEOUT	SDO protocol timed out
0504 0001h	CO_E_SDO_CMD_SPEC_INVALID	client/server command specifier not valid or unknown
0504 0002h	CO_E_SDO_INVALID_BLKSIZE	invalid block size (block mode only)
0504 0004h	CO_E_SDO_INVALID_BLKCRC	CRC error (block mode only)
0504 0005h	CO_E_MEM	out of memory
0601 0001h	CO_E_NO_READ_PERM	attempt to read a write only object
0601 0002h	CO_E_NO_WRITE_PERM	attempt to write a read only object
0602 0000h	CO_E_NONEXIST_OBJECT	object does not exist in the object dictionary
0604 0041h	CO_E_MAP	object can not be mapped to the PDO

SDO Abort Code	RET_T Value	Description
0604 0042h	CO_E_DATA_LENGTH	the number and length of the objects to be mapped would exceed PDO length
0604 0043h	CO_E_PARA_INCOMP	general parameter incompatibility reason
0606 0000h	CO_E_HARDWARE_FAULT	access failed due to an hardware error
0607 0010h	CO_E_WRONG_SIZE	data type does not match, length of service parameter does not match
0609 0011h	CO_E_NONEXIST_SUBINDEX	sub-index does not exist
0609 0030h	CO_E_TRANS_TYPE	value range of parameter exceeded (only for write access)
0609 0031h	CO_E_VALUE_TO_HIGH	value of parameter written too high
0609 0032h	CO_E_VALUE_TO_LOW	value of parameter written too low
060A 0023h	CO_E_SRD_NO_RESSOURCE	no resources available
0800 0000h	CO_E_SDO_OTHER	general error
0800 0020h	CO_E_INVALID_TRANSMODE	data can not be transferred or stored to the application
0800 0022h	CO_E_DEVICE_STATE	data can not be transferred or stored to the application because of the present device state
0800 0023h	CO_E_SRD_NO_RESSOURCE	object dictionary dynamic generation fails or no object dictionary is present (e.g. object dictionary is generated from file and generation fails because of an file error)
0800 0024h	CO_E_NO_DATA_AVAILABLE	no data available

Table 56, SDO Abort Codes

10.4. Appendix — Tools

In the context of the CANopen Library there are many useful tools available for software implementation and system integration.

All tools are available via the web page of *port* . ⟨http://www.port.de⟩ Without a valid license file the tools can be used in demo mode.

10.4.1. CANopen Design Tool

The *CANopen Design Tool* makes the configuration of the CANopen Library and the creation of the object dictionary possible. This tool generates source code file in C for the CANopen Library, electronic device descriptions and various documentations.

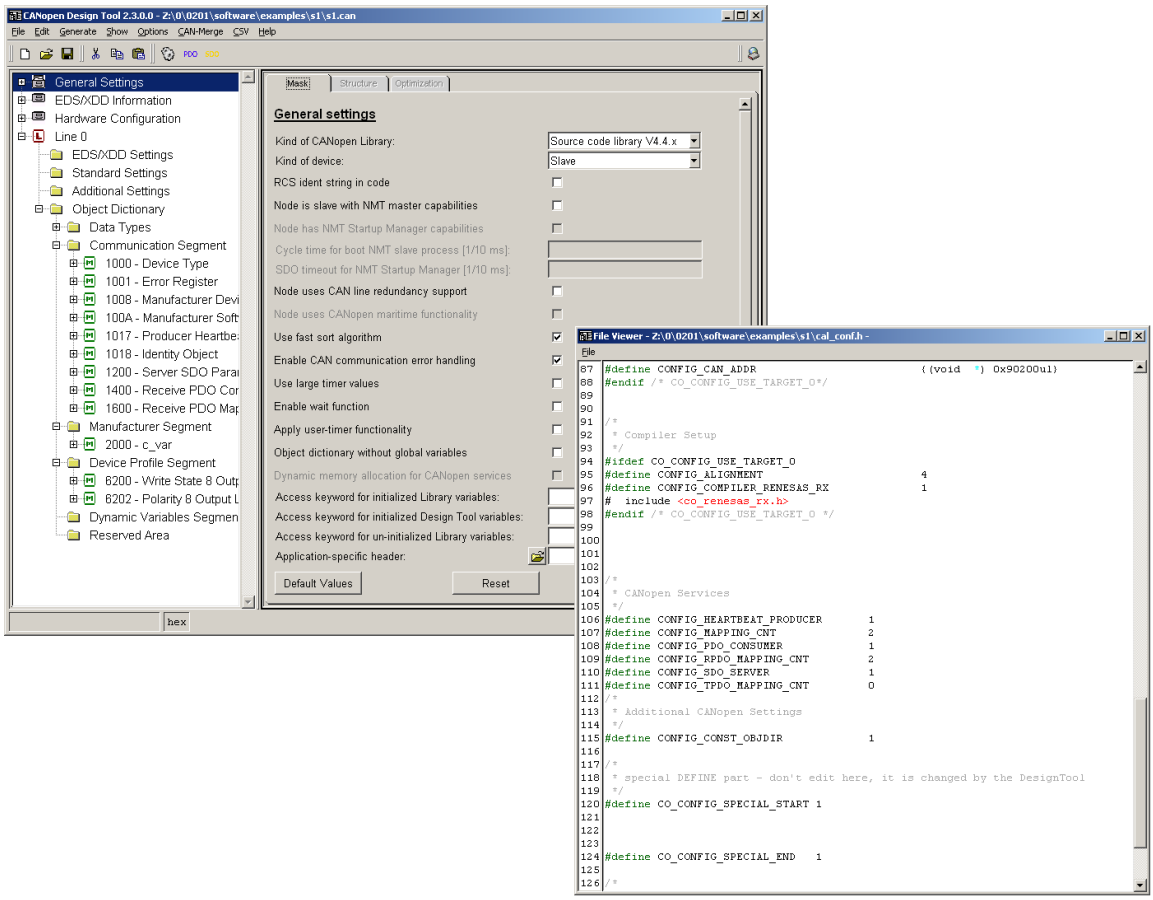


Figure 58, *CANopen Design Tool*

10.4.2. CANopen Server

The CANopen Server is one of the more comprehensive examples for the use of the CANopen Library. The CANopen Server realizes a complete Class 3 CANopen manager node according to the CiA specification CiA-309-3.

With its additional console interface it is possible to test one's own application.

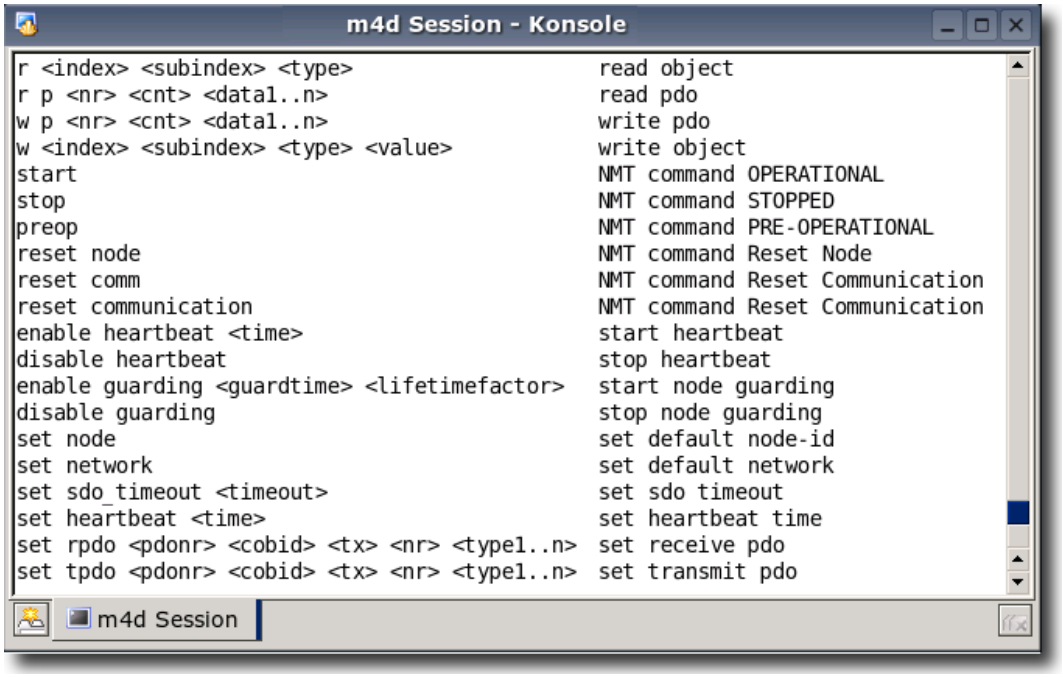


Figure 59, CANopen CiA-309-3 Gateway (m4d)

A commercial full featured version of **m4d** is available. It provides a complete set of CANopen services as well as an easy to use server interface to access all services via TCP/IP sockets.

10.4.3. CANopen Device Monitor

The *CANopen Device Monitor* of *port* is a tool for the graphical inspection and configuration of CANopen devices in a CANopen network. The embedded scripting ability makes it possible to access the implemented CANopen services and to write test or control applications with a minimum of effort.

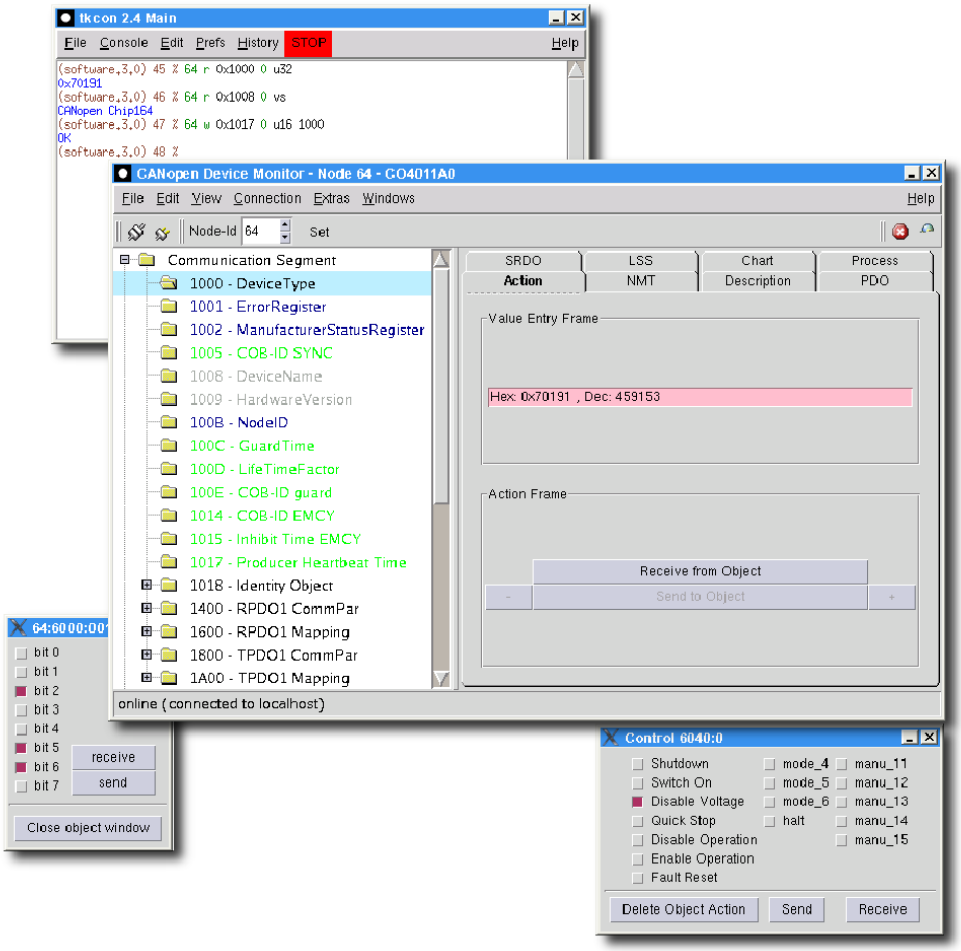


Figure 60, *CANopen Device Monitor*

10.4.4. CAN-REport

The CAN-RE***port*** is a monitoring tool for CAN traffic. It is useful for message recording and interpretation.

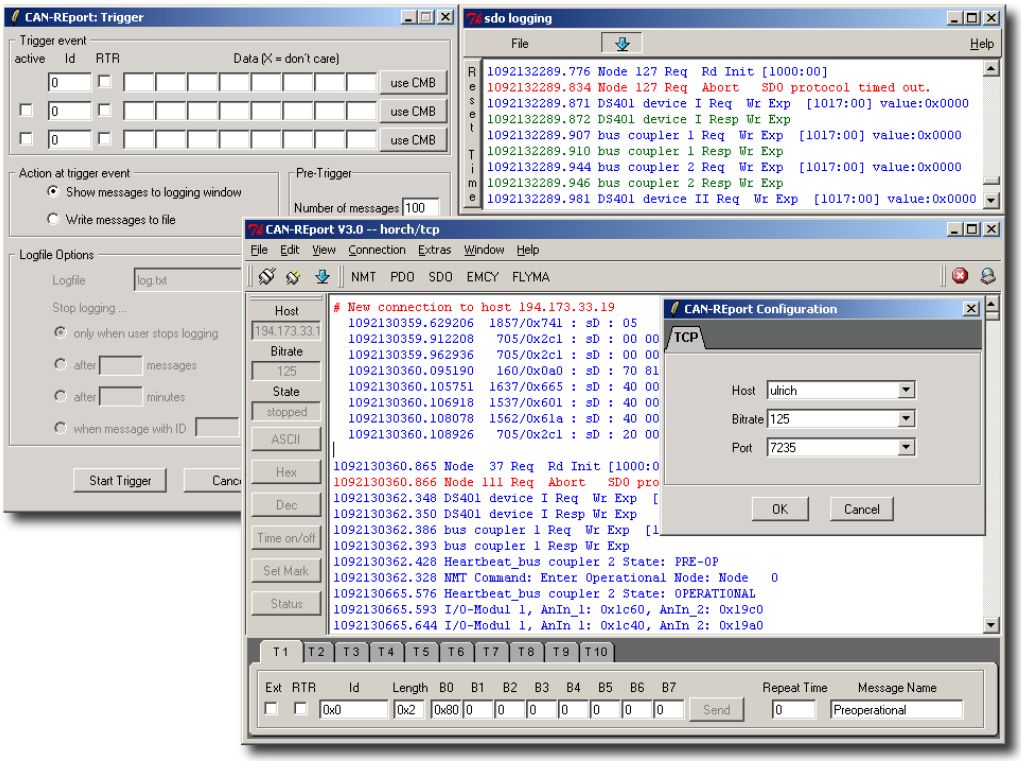


Figure 61, CAN-RE***port***

10.5. Appendix — Abbreviations

CAN	Controller Area Network
CAL	CAN Application Layer (CANopen base)
CDM	*CANopen Device Monitor*
CDT	*CANopen Design Tool*
CiA	CAN in Automation international users and manufacturers group e.V.
CMS	CAN Message Specification
COB	Communication Object (CAN Message)

COB-ID	Communication Object Identifier
CRC	Cyclic redundancy check. Bit error protection method for data communication.
CSDO	
DAM	Destination Address Mode
EDS File	Electronic Data Sheet. This is a file with the device specific parameter description and is provided by the manufacturer of a DeviceNet or CANopen device.
EMCY	Emergency Object
HAL	Hardware Abstraction Layer
HMI	Human Machine Interface
IG	Interest Group, working group within CiA responsible for higher layer protocols, e.g. CANopen
ISR	Interrupt Service Routine
LME	Layer Management Entity
LMT	Layer Management
LSS	Layer Setting Services
MPDO	Multiplexed PDO
NMT	Network Management
OD	Object Dictionary
PDO	Process Data Objects. They are messages in a unconfirmed service. They are used for the transfer of real-time data to and from the device.
RPDO	Receive PDO
RTR	Remote Transmission Request
SAM	Source Address Mode
SCT	Safeguard Cycle Time (CANopen Safety)
SDO	Service Data Objects, messages in a confirmed service. They are used for the access of entries in the object dictionary.
SIG	Special Interest Group, working group within CiA
SRD	SDO Requesting Device
SRDO	Safety Relevant Data Objects
SRVT	Safety Relevant object Validation Time (CANopen Safety)
SSDO	server SDO
SYNC	Synchronization Object (CANopen communication object)
TIME	Time Stamp Object
TPDO	Transmit PDO

10.6. Appendix — Modification for Version V4.x

The chapters describing the modifications made against the previous one. This should help upgrading older projects to the latest CANopen Library version.

10.6.1. Modification Summary V4.5

- Adaption to actual standard CiA-301, V 4.2 (June 2011).
- The CANopen Library supports optional object function pointer now.
- The object attributes got changed from UNSIGNED8 to UNSIGNED16. This is due the additional attribute flags needed now and in the future.
- A new attribute for signaling non volatile storage is added to the object description, to ease the selection which object should be stored.
- New request getObjStoreEnabledReq, which returns if an object should be stored in non volatile memory.
- For speed optimization the access function got changed. This will enhance the startup time and SDO services.
- The CANopen Library discards SDO messages which where not requested or do not have an multiplexer.

10.6.2. Modification Summary V4.4

- Adaption to actual standard CiA-301, V 4.1 (October 2006)
- All CANopen Library and driver internal structures are static now, no *malloc*() calls necessary.
- All driver functions return a RET_T return value. It is evaluated by the CANopen Library.
- optimization of memory usage for internal structures
- fast sort algorithm (for large data arrays) available
- no hardware dependencies in the CANopen Library (no full/basic can checks more) (RTRs are always answered by the CANopen Library)
- CANopen Library supports 29 bit identifier - usage is depending on the driver
- NMT master, Node Guarding master and Heartbeat consumer can be used independent. But they can be used also with the function *addRemoteNodeReq()*.
- CANopen Library status flags are divided into `coLibFlags` and `coCanFlags`. The CAN flags shows at all times the actual state of the driver.
- The boot-up messages can be received from each node, if it is initialized for Node Guarding master or Heartbeat consumer.

- If the emergency list at the object dictionary is available, all nodes from the list will be initialized for EMCY consumer. If the list is not present, nodes can be used as EMCY consumer by calling *setEmcyConsumerCobId(node,id)*.
- SDO client transfers are all times finished by a confirmation function. Time out monitoring and transmission of an abort is automatically done by the CANopen Library.
- Indication/confirmation function for SDO transfers after determined count of data.
- Start/stop SYNC set also the bits for the SYNC-COB-ID at the OD.

10.6.2.1. Modification of the User Interface

- All driver functions returns a RET_T value.
- *defineHBConsumer()* - initialize the Heartbeat consumer list
- *setEmcyConsumerCobId(node,id)* initialization of EMCY consumer, if the EMCY consumer list at the object dictionary does not exist.
- *defineEmcy(PRODUCER/CONSUMER)*
- *writeEmcyReq()*
- *emcyInd(node,emcy_t)*
- *addGuardingSlave(node,guardTime,lifeTime)*
- *setGuardTimePara(node,guardTime,lifeTimeFac)*
- *getRemoteNodeState()*
- *createNetworkReq()*
- *addRemoteNodeReq(node,GuardTime,LifeTime,useHB,useGuard)*
- *writeSdoReq(sdoNr,index,subIndex,*pData,length,timeOut))*
- *readSdoReq(sdoNr,index,subIndex,*pData,length,timeOut))*
- *rtrPdoInd(pdoNr)*
- *waitForSdoRes(sdoNr)*
- *setCobId(index,subIndex,cobid)*

10.6.3. Modification Summary V4.3

- new timer concept
- new modules: SRDO, LSS, LED
- new NMT function setNodePREOP() - set node in state PRE-OPERATIONAL
- documentation revision
- virtual objects introduced
- defines for COB-IDs unified in module **co_cobid.h**

- Type of COB passed to the driver
- defines for SDO command codes introduced
- coWait not available by default (enable with define)
- addRemoteNodeReq not necessary for local node
- ResetComm changes values only in object dictionary to the defaults Other values are reloaded with *loadParameterInd()*. This values are then accounted when setting the COB-IDs
  The same applies for the start of the program: with *initCANopen()* the object dictionary is initialized. After that the user can change the entries in the object dictionary with *loadParaInd()*. These changes are accounted when the services are initialized.

10.6.3.1. Modification of the User Interface

For unified function names the following function names has been changed:

old function name	new function name
resetApplication()	resetApplInd()
resetCommunication()	resetCommInd()

For optimal support some function parameter or return values have been changed:

- Return values of sdoWrInd()/sdoRdInd changed - possible values are:
  CO_OK - access ok
  CO_WAIT - indication not finished yet
  CO_xxx - SDO Abort Code
- generic driver: Version for first tests with the new CANopen Library
- driver concept of v4.2 slightly adapted
  Within the driver modules the name of some files has changed:

CANopen Library v4.2	CANopen Library v4.3
can82527.h can82527.c	can_82527.h can_82527.c
can90540.h can90540.c	can_90540.h can_90540.c

10.6.3.2. Changes in the Naming of Configuration Constants

CANopen Library v4.2	CANopen Library v4.3
`CONFIG_STANDARD_xxx`	`CONFIG_COLIB_xxx`
`CAN_ACCESS_xxx`	`CONFIG_CAN_ACCESS_xxx`

CANopen Library v4.2	CANopen Library v4.3
CONFIG_DRIVER_USE_xxx	CONFIG_CAN_USE_xxx
CAN_T_CLK SIZE_POOL CONFIG_SYSTEM_MALLOC[14]	CONFIG_CAN_T_CLK CONFIG_SIZE_POOL CONFIG_COLIB_MALLOC
CO_TIMER_INC CAN_REGISTER_OFFSET CONFIG_GROUP_CHANNEL	CONFIG_TIMER_INC CONFIG_CAN_REGISTER_OFFSET CONFIG_SPECIAL_CHANNEL
TARGET_xxx	CONFIG_CPU_FAMILY_xxx CONFIG_CPU_TYPE_xxx CONFIG_CAN_FAMILY_xxx CONFIG_CAN_TYPE_xxx
CONFIG_CPU_xxx	CONFIG_CPU_FAMILY_xxx CONFIG_CPU_TYPE_xxx
CONFIG_CAN_xxx	CONFIG_CAN_FAMILY_xxx CONFIG_CAN_TYPE_xxx
CAN_ISR_NUMBER_xxx CAN_ISR_REGISTERBANK_xxx	CONFIG_CAN_ISR_NUMBER CONFIG_CAN_ISR_REGISTERBANK
TIMER_ISR_REGISTERBANK TIMER_ISR_NUMBER	CONFIG_TIMER_ISR_REGISTERBANK CONFIG_TIMER_ISR_NUMBER
CAN_C167CS_IPC CAN_IF_REG_VAL CAN_BUS_CONF_VAL CAN_CDR_VAL	CONFIG_CAN_C167CS_IPC CONFIG_CAN_82527_IF_REG_VAL CONFIG_CAN_82527_BUS_CONF_VAL CONFIG_CAN_82C200_CDR_VAL
CONFIG_CPU_F2MC16LX_IRQ_LEVEL_TIMER CONFIG_CPU_F2MC16LX_IRQ_LEVEL_CAN	CONFIG_TIMER_IRQ_LEVEL CONFIG_CAN_IRQ_LEVEL

10.6.4. Modification Summary V4.2

- New header- and c-file concept - divide all functionality depending the services
- New indications added for boot-up and start Heartbeat
- If emergency is supported, the emergency entry in the object dictionary is mandatory
- Inhibit Time for emergency supported
- Multiplexed PDO for destination address mode and source address mode available

[13] The define CONFIG_SYSTEM_MALLOC enables the compiler specific memory allocation.
The define CONFIG_COLIB_MALLOC enables our common customized simply memory allocation.

- New driver for Fujitsu 90540 CPU and CAN controller available
- Support for the TMS320 signal processor family
- Flying Master and Redundancy Support
- Slave with master capabilities available
- New indication function *syncCmd()* to start user function after the received SYNC message

10.6.4.1. Modification of the User Interface

All defines in the CANopen Library are changed to CANopen compatible names

old define	new define
CONFIG_EMCY_SERVER	CONFIG_EMCY_PRODUCER
CONFIG_EMCY_CLIENT	CONFIG_EMCY_CONSUMER
CONFIG_PDO_CLIENT	CONFIG_PDO_CONSUMER
CONFIG_PDO_SERVER	CONFIG_PDO_PRODUCER

10.6.5. Modification Summary V4.1

- Function calls *Enable_CAL_Interrupts*() and *Disable_CAL_Interrupts*() replaced by the macros ENABLE_CAN_INTERRUPTS and DISABLE_CAN_INTERRUPTS
- Using of FLOAT variable type in the object dictionary is now possible
- Define **DATA** replaced by **CO_DATA**
- Prepared more drivers for using multi-line
- Change the driver interface for mod167 to use multiple CAN controller types
- support for SDO block up/download released
  automatic usage by controlling the data size
  if the client was not understand SDO block transfer automatically falls back to segmented transfer
- Optimization for the Heartbeat consumer
- Release SDO Manager/SDO Requester functionality

10.6.5.1. Modification of the User Interface

SDO transfer for block transfer adapted - function *waitForSdoRes()* and abort condition for time out changed

10.6.5.2. Driver Interface

- New macros defined for ENABLE_CAN_INTERRUPTS and DISABLE_CAN_INTERRUPTS for calling *Enable_CAN_Interrupts()/Disable_CAN_Interrupts()*.

- Driver for mod167 prepared for usage with multi can controller types

10.6.5.3. Structures

Only internal structures have been changed.

Object dictionary layout has changed, *CANopen Design Tool* Version number greater than 13 has to be used.

10.6.5.4. Tools

Add new features (SDO block transfer, multi-CAN controller) to the ConfigTool.

10.6.6. Modification Summary V4.0

- CANopen Library is now compatible with the CANopen Standard CiA-301, V4.0
- All CAL dependencies are removed
- All interrupt functions are modified
  CANopen functionality is no longer carried out within the interrupt routines. Within the ISR only flags are set. These flags are valued by the function FlushMbox() and the corresponding function is called. This is why you have to call the function FlushMbox() regularly in the main loop.
- Some of the user functions have been changed.
- The object dictionary can be stored in ROM.
- A lot of the structures have been optimized.
- For a better understanding some of the defines have changed. (There are no longer defines with `_NO_`.)

10.6.6.1. Modification of the User Interface

Please have a look at the reference manual for a detailed description of the functions.

old function	new function
LME_Init_CAL_req("", "", "") LME_Leave_CAL_req()	initCANopen() leaveCANopen()
NMT_CreateNode_req("", lNodeId, 0) NMT_DeleteNode_req()	createNodeReq(CO_TRUE, CO_FALSE) deleteNodeReq()
NMT_CreateNetwork_req(0) NMT_DeleteNetwork_req()	createNetworkReq(CO_FALSE) deleteNetworkReq()
NMT_AddRemoteNode_req("", rNodeId, 1792+rNodeId, 2000, 3) NMT_RemoveRemoteNode_req("", rNodeId)	addRemoteNodeReq(rNodeId, 2000, 3) removeRemoteNodeReq(rNodeId)
definePdo(TRANSMIT_PDO, 1, 0, 0)	definePdo(TRANSMIT_PDO, 1, CO_TRUE)
defineEmcy(CLIENT, &emcy[0], REQ_ID_EMCY1+rNodeId)	defineEmcy(CLIENT, 1, REQ_ID_EMCY1+rNodeId)

10.6.6.2. Modification of the Object Dictionary

The following entries are obsolete:

Index	Sub-Index	Meaning
1004	0	Number of PDOs
100B	0	node-ID
100E	0	COB-ID Guarding
100F	0	Number of SDOs
1400	4	PDO priority
1800	4	PDO priority

New entries at the Object Dictionary:

Index	Sub-Index	Meaning
1016	0-255	Heartbeat consumer Time
1017	0	Heartbeat producer Time
1018	0..4	Identity Object
1800	5	Event Time

- The object dictionary now has the new type **OBJDIR_T**.
- With the define `CONFIG_CONST_OBJDIR` it can be saved in ROM.

10.6.6.3. Structures

The following structures have been optimized (especially for byte alignment on 8 bit controllers):

```
typedef struct
{
    UNSIGNED8    *pObj;          /* pointer to data */
    VALUE_DESC_T *pValDesc;      /* value description */
    UNSIGNED16   index;          /* index of object */
    UNSIGNED8    numOfElem;      /* number of elements */
} LIST_ELEMENT_T;

typedef struct
{
    UNSIGNED32  defaultVal;        /* default value or size of domains */
#ifdef CONFIG_LIMITS_CHECK
    UNSIGNED32  minRange;          /* min. range of object element */
    UNSIGNED32  maxRange;          /* max. range of object element*/
#endif
    INTEGER8    size;              /* size of element in Bytes
                                      if size negative object is value
                                      a signed type*/
    UNSIGNED8   attribute;         /* domain type = 1, short desc = 2,
                                      reserved  = 4, num_val = 0x10,
                                      read permitted = 0x20,
                                      write permitted = 0x40,
                                      write permitted = 0x40,
                                      pdoMAPPING allowed = 0x80 bit-coded ! */
} VALUE_DESC_T;
```

11. Index

- V -

- W -