www.em-sa.com/nxp

EmSA CANopen (FD) Libraries for NXP SDKs

User Manual

Based on Version 7.01 of Micro CANopen Plus

1st of June 2020

Software License

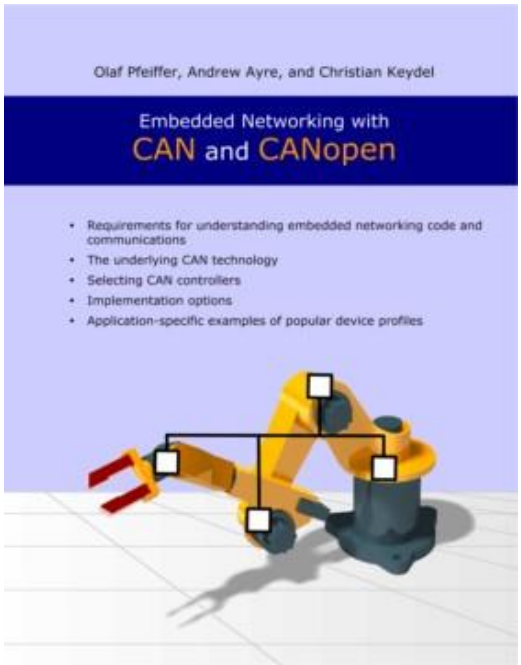
Contents

1 About CANopen (FD)

CANopen and CANopen FD are CAN (Controller Area Network, standardized in ISO 11898 series) protocols usable for almost any type of application. One of the primary uses cases is “embedded machine control”, so any machinery handling materials or moving, driving, swimming, flying. The CANopen and CANopen FD specifications are maintained by the CAN in Automation (CiA) international user’s and manufacturer’s group (www.can-cia.org).

1.1 CANopen Documentation and Terms

It is assumed that programmers using the CANopen libraries have a general understanding about how CANopen works. In addition, they should either have access to the CANopen specifications or a CANopen book such as “Embedded Networking with CAN and CANopen” (www.CANopenBook.com). Another source for information is our video channel at www.em-sa.com/video.



This manual does not explain regular CANopen features, functions and terms other than the ones below:

- **Node ID:**
  each CANopen device physically connected to CAN requires a unique node ID in the range of 1 to 127 and is then referred to as a CANopen node.
- **Object Dictionary:**
  each CANopen node provides an Object Dictionary (OD). Single entries (parameters) in the OD are addressed using an Index and Subindex

value. In summary, the OD is a list of all parameters that the node can communicate via CANopen.

- **Process Image:**
  in the CANopen libraries, all OD process data communicated is stored in a process image. Individual entries are addressed using an offset value. Here process image layout and offsets are auto-generated by the CANopen Architect EDS editor.
- **EDS:**
  Electronic Data Sheets document the implemented OD and other configuration of a CANopen device.
- **XDD:**
  XML-based eXtensible Device Description file format to replace the EDS format in the future. Mandatory to use for CANopen FD devices as EDS lacks support for CANopen FD.
- **NMT State:**
  a CANopen node implements a network management state machine. The states available in CANopen (FD) are Boot, Pre-Operational, Operational, Stopped.
- **NMT Master:**
  the network management master sets the NMT state of nodes by transmitting the NMT Master message.
- **Bootup/Heartbeat:**
  each CANopen node produces a bootup and a cyclic heartbeat message. The message content is the current NMT state of the node.
- **EMCY:**
  CANopen nodes can produce emergency messages to inform others of errors or alarms.
- **SDO:**
  Service Data Object (available in CANopen) is a request and response communication service to access entries in the Object Dictionary of a node.
- **USDO:**
  the Universal Service Data Object (available in CANopen FD) is an extended request and response communication service to access entries in the Object Dictionary of one or multiple nodes.
- **Client (SDO or USDO):**
  the (U)SDO client sends an OD read or write request to one or multiple (U)SDO server.
- **Server (SDO or USDO):**
  the (U)SDO server sends responses to OD read or write requests received from clients. It "serves" the data from its own OD to the network.
- **PDO:**
  Process Data Objects are application or time triggered messages containing the data from one or multiple OD entries.

1.2 CANopen and CANopen FD

The original specifications for the protocols are available at no cost through the CAN in Automation (CiA) international user's and manufacturer's group (www.can-cia.org).

- For CANopen, refer to document CiA 301 "CANopen application layer and communication profile" version 4.2.0.
- For CANopen FD, refer to CiA 1301 "CANopen FD Application Layer and Communication Profile" version 1.00.
- For CANopen Manager functionality, refer to CiA 302-x documents like CiA 302-2 "CANopen additional application layer functions – Network Management".
- For Device and Application profiles, refer to CiA 4xx documents such as CiA 401 "CANopen device profile for generic I/O modules".

1.3 CANopen Devices and CANopen Manager

Our implementations support both a CANopen Device (sometimes still referred to as a "slave") and the CANopen Manager. The CANopen Manager is typically used by a control application (controlling all the connected devices) and always includes a CANopen Device, too.

2 CANopen (FD) in the NXP SDK

The CANopen Libraries for this distribution are provided by Embedded Systems Academy and are based on their Micro CANopen Plus implementation version 7.0. See www.em-sa.com/nxp for more information.

2.1 Release notes

This manual is for the initial release prepared on 2nd of April 2020.

See www.em-sa.com/nxp for a list of available releases.

2.2 The License Summary

See page 2 of this manual for the complete license text. In short: the libraries are free to use with the NXP devices they are published for. Do not modify the library in any way.

2.3 CANopen vs CANopen FD

Our implementations support both classical CANopen and CANopen FD. Select the appropriate library (name indicates "CANopen" or "CANopenFD") to either choose classical CANopen or CANopen FD.

2.4 CANopen Devices and CANopen Manager

Select the appropriate library (name indicates "Device" or "Manager") to either choose plain CANopen Device or Manager functionality.

2.4.1 Features of the CANopen Device Library

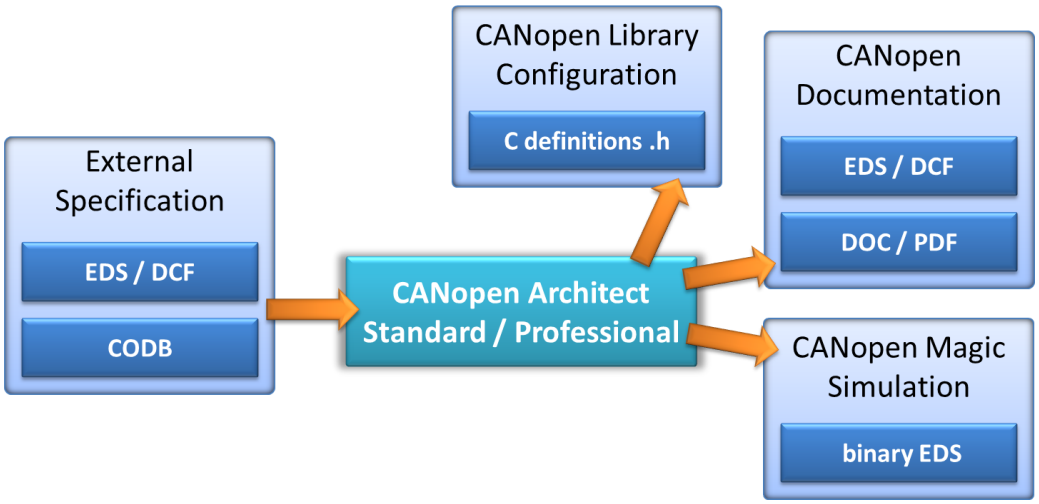
- Node ID must be in range from 1 to 10
- Configurable Object Dictionary to allow support of most Device and Application Profiles
- Bootup and Heartbeat producer
- EMCY producer
- (U)SDO server
- Maximum of 8 PDOs (4 RPDO / 4 TPDO)

2.4.2 Features of the CANopen Manager Library (incl. Device Library)

- Handling of up to 10 nodes (node id of devices must be in range of 1 to 10)
- Monitoring Bootups, Heartbeats and NMT States of all nodes
- (U)SDO client access to nodes (send read/write requests to any OD entry)
- Autoscan of multiple entries after device detection
- EMCY consumer
- Maximum of 32 PDOs (16 RPDO / 16 TPDO)

2.5 Object Dictionary Configurability

The Object Dictionary contents can be freely configured using EmSA's CANopen Architect Standard EDS editor available from www.em-sa.com/nxp. The editor can be used to create or modify the Object Dictionary used by a CANopen (FD) node. A configuration can be saved as EDS but also exported for inclusion in the CANopen libraries. The option "Export Micro CANopen Plus Sources…" generates the C definition .h files used by the libraries.



The optional professional version offers extended features like documentation export and advanced console commands to manage a large number of devices, entries and PDOs.

2.6 Limitations

The CANopen (FD) libraries provided with the NXP SDK do not implement all CANopen (FD) features and functions available. The functionality included is suitable for simple I/O devices and minimal control applications that can pass the official CANopen Conformance Test. The limitations are shown in the table below.

	Device Library	Manager Library	Commercial libraries from EmSA
Node ID selection	1 to 10	1 to 10	1 to 127
Transmit PDO	0 to 4	0 to 16	0 to 512
Receive PDO	0 to 4	0 to 16	0 to 512
OD entries	virtually unlimited	virtually unlimited	virtually unlimited
Maximum nr of nodes handled (Manager)	n.a.	10	127

Extended functionality, commercial libraries are available from EmSA, see www.em-sa.com/nxp.

2.7 Extended testing and debugging

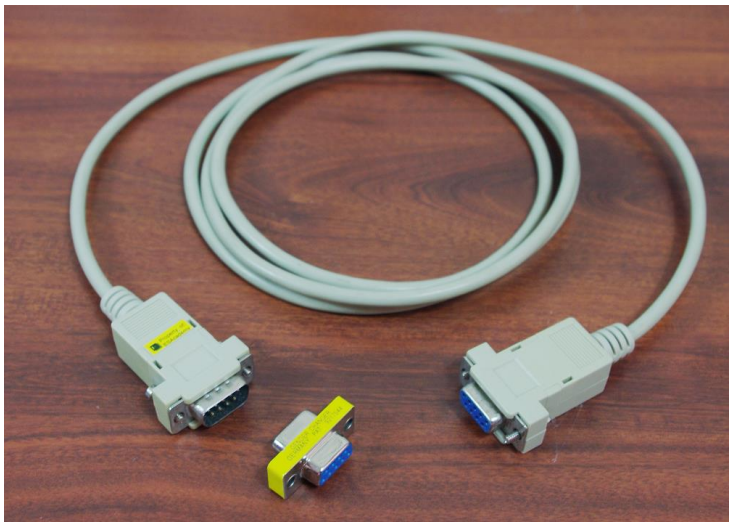
For test and analysis of the CANopen (FD) communication any CANopen (FD) monitoring, configuration or analysis tools like www.canopenmagic.com can be used.

Extended functionality libraries are available from EmSA, see www.em-sa.com/nxp for details.

3 Getting started, step-by-step

This chapter describes the steps involved to operate the CANopen examples provided.

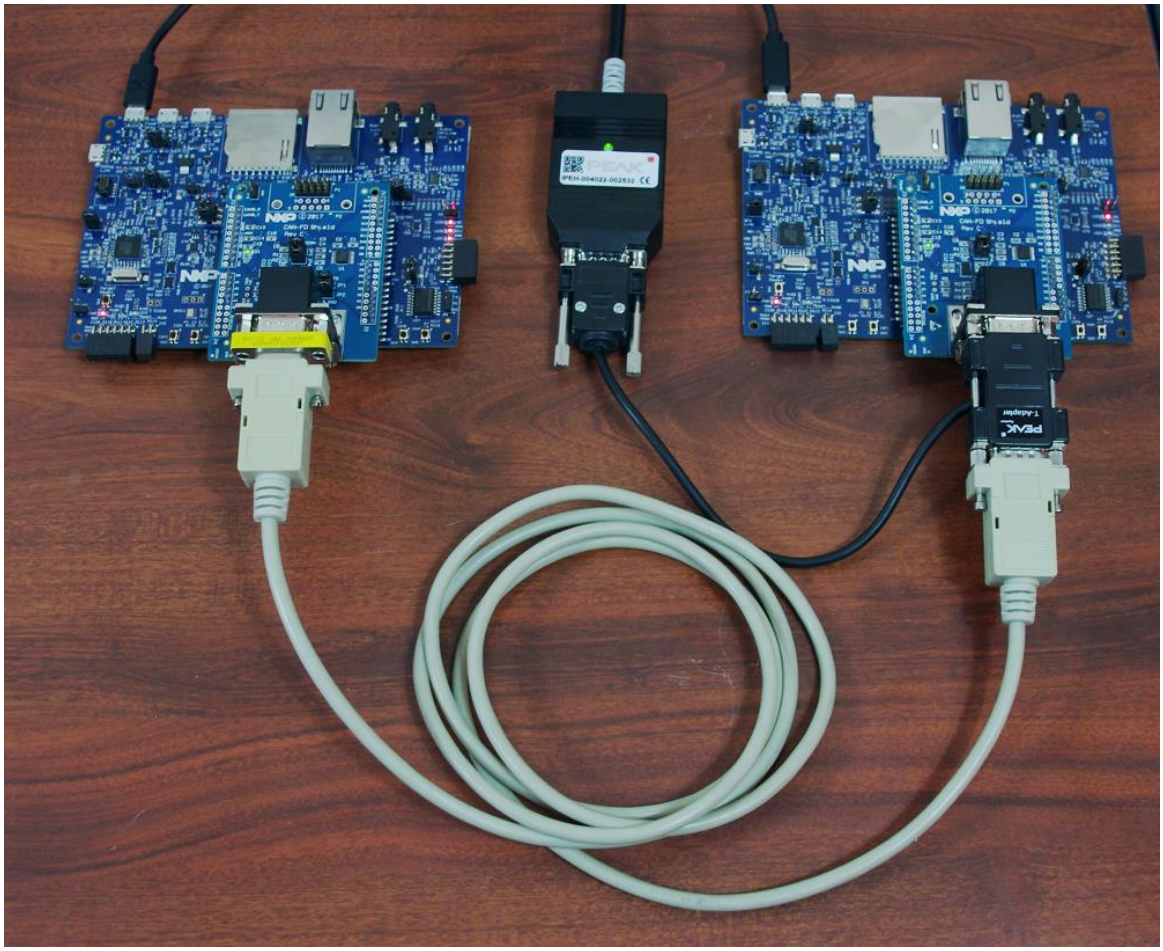
3.1 Hardware setup



You need two boards supported by the CANopen libraries to operate this example. One board is programmed with the CANopen Device example, the other one is programmed with the CANopen Manager example. Note that some boards require an optional CAN FD shield to provide the CAN connector. See www.em-sa.com/nxp for a list of supported boards.

The CAN DB9 ports used must be properly connected with CAN cabling, as a minimum the pins CAN_L, CAN_H and GND need to be connected. Often RS232 cables like shown above can be used with gender changers. (NO Null-Modem cable!).

Termination resistors are required at both ends of the network. Many evaluation boards have these already on-board, typically jumpers are used to enable/disable them.



The setup in the previous picture consist of one LPCXpresso54628 and LPCxpresso54S018 each with a CAN shield and termination resistors enabled. A RS232 cable with one gender changer, a 'Y' (or'T') cable and a PCAN-USB FD to monitor the CANopen communication.

3.2 Network Monitoring

It is highly recommended to use a CAN, or better CANopen monitoring and analysis tool (like www.canopenmagic.com) also connected to the network. This allows you to monitor the CANopen traffic exchanged by the two nodes and transmit further interactive test messages.

The default bitrate of the example sis 500kbps.

3.3 Loading the Software

The CANopen library examples can be installed and programmed in the same manner as other NXP SDK program examples. Simply select the evaluation board and the CANopen device or manger example and use your preferred tool chain to program the software into the flash memory of the boards.

One board needs the CANopen Device example programmed, the second the CANopen Manager example.

Initially use the debug versions of the libraries, these provide outputs to the debug console.

3.4 Running the examples

If you power-up or start both devices at the same time and have a CANopen monitor connected, then you will see CANopen messages exchanged as shown in the following table.

Msg Nr	ID	Message Type	Node	Details	Data (Hex), Comment
1	0x703	Bootup	0x03 (3)		
2	0x083	Emergency	0x03 (3)	[0x0000 (0)] Error reset or no error	00 00 00 00 00
3	0x701	Bootup	0x01 (1)		
4	0x081	Emergency	0x01 (1)	[0x0000 (0)] Error reset or no error	00 00 00 00 00
5	0x701	Heartbeat/Node Guarding	0x01 (1)	Operational	
6	0x000	NMT Master Request	All	Reset	MGR: reset all nodes
7	0x703	Bootup	0x03 (3)		
8	0x000	NMT Master Request	0x03 (3)	Pre-Operational	
9	0x083	Emergency	0x03 (3)	[0x0000 (0)] Error reset or no error	00 00 00 00 00
10	0x603	SDO Download Request	0x03 (3)	[0x1017,0x00] Producer HB Time, exp.	EE 02
11	0x583	SDO Download Response	0x03 (3)		Write 750ms
12	0x603	SDO Download Request	0x03 (3)	[0x1016,0x01] Consumer HB Time 1, exp.	E8 03 01 00
13	0x583	SDO Download Response	0x03 (3)		Write 1000ms
14	0x701	Heartbeat/Node Guarding	0x01 (1)	Operational	
15	0x603	SDO Initiate Upload Request	0x03 (3)	[0x1000,0x00] Device Type	Read device type
16	0x583	SDO Upload Response	0x03 (3)	91 01 00 00, expedited	91 01 00 00
17	0x603	SDO Initiate Upload Request	0x03 (3)	[0x1018,0x01] Identity - Vendor ID	Read vendor id
18	0x583	SDO Upload Response	0x03 (3)	DC 02 00 AF, expedited	DC 02 00 AF
19	0x603	SDO Initiate Upload Request	0x03 (3)	[0x1018,0x02] Identity - Product Code	Read product code
20	0x583	SDO Upload Response	0x03 (3)	10 00 DE C0, expedited	10 00 DE C0
21	0x000	NMT Master Request	0x03 (3)	Operational	
22	0x183	Default: PDO		Default: TPDO 1 of Node 0x03 (3)	04 02 00 00
23	0x283	Default: PDO		Default: TPDO 2 of Node 0x03 (3)	01 00 00 00
24	0x183	Default: PDO		Default: TPDO 1 of Node 0x03 (3)	04 02 00 00
25	0x701	Heartbeat/Node Guarding	0x01 (1)	Operational	
26	0x283	Default: PDO		Default: TPDO 2 of Node 0x03 (3)	02 00 00 00
27	0x183	Default: PDO		Default: TPDO 1 of Node 0x03 (3)	04 02 00 00
28	0x183	Default: PDO		Default: TPDO 1 of Node 0x03 (3)	04 02 00 00
29	0x183	Default: PDO		Default: TPDO 1 of Node 0x03 (3)	04 02 00 00
30	0x283	Default: PDO		Default: TPDO 2 of Node 0x03 (3)	03 00 00 00
31	0x701	Heartbeat/Node Guarding	0x01 (1)	Operational	
32	0x703	Heartbeat/Node Guarding	0x03 (3)	Operational	
33	0x603	SDO Initiate Upload Request	0x03 (3)	[0x1008,0x00] Manuf. Device Name	
34	0x303	Default: PDO		Default: RPDO 2 of Node 0x03 (3)	01 01 00 00
35	0x203	Default: PDO		Default: RPDO 1 of Node 0x03 (3)	00 00 00 00

36	0x583	SDO Upload Response	0x03 (3)	fragmented, size = 31 bytes	
37	0x603	SDO Upload Segment Req	0x03 (3)	toggle = 0	
38	0x583	SDO Upload Response	0x03 (3)	toggle = 0, C A N o p e n	43 41 4E 6F 70 65 6E
39	0x603	SDO Upload Segment Req	0x03 (3)	toggle = 1	
40	0x583	SDO Upload Response	0x03 (3)	toggle = 1, L i b N X P S	4C 69 62 4E 58 50 53
41	0x603	SDO Upload Segment Req	0x03 (3)	toggle = 0	
42	0x583	SDO Upload Response	0x03 (3)	toggle = 0, D K C i A 4	44 4B 20 43 69 41 34
43	0x603	SDO Upload Segment Req	0x03 (3)	toggle = 1	
44	0x583	SDO Upload Response	0x03 (3)	toggle = 1, 0 1 E x a m	30 31 20 45 78 61 6D
45	0x603	SDO Upload Segment Req	0x03 (3)	toggle = 0	
46	0x583	SDO Upload Response	0x03 (3)	toggle = 0, last segment	70 6C 65
47	0x183	Default: PDO		Default: TPDO 1 of Node 0x03 (3)	04 03 00 00
48	0x203	Default: PDO		Default: RPDO 1 of Node 0x03 (3)	00 00 00 04
49	0x183	Default: PDO		Default: TPDO 1 of Node 0x03 (3)	05 03 00 04
50	0x203	Default: PDO		Default: RPDO 1 of Node 0x03 (3)	00 00 00 05

Line 1 to 4: On startup, the Manager (Node 1) and the Device (Node 3), transmit their bootup messages and an emergency reset message (clears all potentially pending emergencies).

Line 6: Manager requests that all nodes reset (ensures that Manager sees also bootups of nodes potentially already running).

Line 8: Manager requests, that node 3 remains in pre-operational mode, so that it can get scanned and configured.

Line 10, 12, 15, 17, 19: Manager performs a configurable autoscan of node found, here node 3. All devices found get configured with a heartbeat producer and consumer time (listening to heartbeat of Manager) and device information is read. In all cases node 3 sends the appropriate SDO response as a confirmation that the service completed.

Line 21: Manager requests, that node 3 goes into operational mode and starts handling PDOs.

Line 22 and next: Both devices start transmitting PDOs. RPDOs are generated by the Manager and consumed by node 3.

Line 33 and following: Manager initiates a segmented SDO transfer, reading the manufacturer device name, here 31 bytes transferred in multiple segments.

The expected debug console output of the Manager is shown below (start Manager first, then device):

```
Starting CANopen FD Library manager example
Provided by EmSA - www.em-sa.com/nxp

CANopen FD Library Event - Reset Communication, nominal bitrate 500kbps, data
bitrate 2000kbps, node id 1
CANopen FD Library Event - NMT Change: 0x 0 boot
CANopen FD Library Event - NMT Change: 0x 5 operational
CANopen FD Manager Event - Node Status Change: 3, 0x 0 booted
CANopen FD Manager Event - Node Status Change: 3, 0x81 emergency over / reset
CANopen FD Manager Event - Node Status Change: 3, 0xA0 scan complete
 [1000,01]:   F0191 [1018,01]:AF0002DC [1018,02]:C0DE003'
CANopen FD Manager Event - Node Status Change: 3, 0x 5 operational
CANopen FD Manager Event - Node Status Change: 3, 0x90 heartbeat monitoring ac-
tive
CANopen FD Manager USDO Client Complete: node 3
```

```
 [1008,00]:CANopenLibNXPSDK CiA401 Example
 [6411,01]:0x    2
 [6411,01]:0x    2
```

 [6411,01]:0x   2 Numbers in "[ ]" indicate an Object Dictionary (16bit Index and 8 bit Subindex, both hexadecimal). Here selected data received from the CANopen node 3. The highlighted values are only displayed, if DEBUG_CONSOLE is defined when building the project.

The expected debug console output of the Device is shown below (started while Manager is running):

```
Starting CANopen {FD} Library slave example
Provided by EmSA - www.em-sa.com/nxp

CANopen FD Library Event - Reset Communication, nominal bitrate 500kbps, data
bitrate 2000kbps, node id 3
CANopen FD Library Event - NMT Change: 0x 0 boot
CANopen FD Library Event - NMT Change: 0x7F pre-operational
CANopen FD Library Event - NMT Change: 0x7F pre-operational
CANopen FD Library Event - NMT Change: 0x 5 operational
 [6200h,01h]:0x00 [6200h,02h]:0x00 [6200h,03h]:0x00 [6200h,04h]:0x00
[6411h,01h]:0x0000 [6411h,02h]:0x0000
 [6411h,01h]:0x0003
 [6200h,03h]:0x04 [6200h,04h]:0x03
 [6200h,03h]:0x06
 [6411h,01h]:0x0000
 [6200h,03h]:0x07
 [6200h,04h]:0x04
 [6411h,02h]:0x0003
```

Highlighted numbers in "[ ]" are only displayed, if DEBUG_CONSOLE is defined. These indicate an Object Dictionary (16bit Index and 8 bit Subindex, both hexadecimal). Here data received from the Manager.

3.5 CANopen Device: generic I/O Example Application

The example code supplied implements a minimal CiA 401-compliant I/O device with 4 digital input bytes, 4 digital output bytes, 2 analog 16bit inputs and 2 analog 16bit outputs. The process data is transmitted using 2 Transmit PDOs and 2 Receive PDOs.

The default bitrate is 500kbps, the default node ID is 3 and the default heartbeat time is 1s.

All process data exchanged is handled in function *USER_Process()*. Triggering of the TPDOs is determined by a combination of change of state (COS) detection and event and inhibit timers. See the EDS or CANopen Architect configuration for detailed PDO configuration.

Receive PDO 1 data (received from Manager, CAN ID 203h) is copied to:

- [6200h,1] Digital out byte 1
- [6200h,2] Digital out byte 2
- [6200h,3] Digital out byte 3
- [6200h,4] Digital out byte 4

Receive PDO 2 data (received from Manager, CAN ID 303h) is copied to:

- [6411h,1] Analog out word16 1
- [6411h,2] Analog out word16 2

Transmit PDO 1 data with CAN ID 183h contains:

- [6000h,1] Digital in byte 1: counter of digital data received
- [6000h,2] Digital in byte 2: counter of analog data received
- [6000h,3] Digital in byte 3: copy/echo of [6200h,3] Digital out byte 3
- [6000h,4] Digital in byte 4: copy/echo of [6200h,4] Digital out byte 4

Transmit PDO 2 data with CAN ID 283h contains:

- [6401h,1] Analog out word16 1: timer with hi byte seconds and lo byte quarter seconds
- [6401h,2] Analog out word16 2: copy/echo of [6411h,2] Analog out word16 2

A Manager can also use additional SDO services to read/write process data. The example manager application cyclically reads [6000h,3] to demonstrate SDO usage.

3.6 CANopen Manager: Control Example Application

The example code supplied implements a minimal Manager. For PDO handling the same matching of the device is used.

The default bitrate is 500kbps, the default node ID is 1 and the default heartbeat time is 333ms.

All process data exchanged via PDO is handled in function *USER_ProcessApp()*. Triggering of the TPDOs is determined by a combination of change of state (COS) detection and event and inhibit timers. See the EDS or CANopen Architect configuration for detailed PDO configuration.

When the first CiA 401 compatible device is found, the Manager modifies its own PDO CAN ID to match those of the device. It consumes the device's TPDOs and produces the device's RPDOs.

Transmit PDO 1 data contains:

- [6000h,1] Digital in byte 1: 0
- [6000h,2] Digital in byte 2: 0
- [6000h,3] Digital in byte 3: copy/echo of [6200h,1] Digital out byte 1
- [6000h,4] Digital in byte 4: copy/echo of [6200h,2] Digital out byte 2

Transmit PDO 2 data contains:

- [6401h,1] Analog out word16 1: timer with hi byte seconds and lo byte quarter seconds
- [6401h,2] Analog out word16 2: copy/echo of [6411h,2] Analog out word16 1

3.7 Classical CANopen vs. CANopen FD

The differences between the classical CANopen and CANopen FD examples are minimal.

- The default bitrate for CANopen FD is a nominal bitrate of 500kbps and a data bitrate of 2000kbps.
- The debug console output indicates "CANopen FD" instead of "CANopen"
- To show the increased data capacity of CANopen FD, the CANopen FD examples use a longer analog data array as follows:
    - The CANopen FD Device example uses 8 instead of 2 analog inputs from [6401h,1] to [6401h,8]. All 8 entries are mapped into Transmit PDO 2 giving it a total length of 16 bytes.
    - The CANopen FD Manager example uses 8 instead of 2 analog outputs from [6411h,1] to [6411h,8]. All 8 entries are mapped into Receive PDO 2 giving it a total length of 16 bytes.

4 Using the CANopen Libraries

4.1 File and Directory Structure

The directory structure used by the CANopen Libraries and examples separates the files used into major groups. It is recommended to maintain this structure and to adopt it for the grouping of source files in the project settings and layouts as supported by most compiler systems.

The path component "./" stands for the project root, as shown in the respective IDE.

4.1.1 Common Shared Directory

Environment	Path
Native/file system	./MCO
MCUXpresso	./mco
Keil uVision	./MCO
IAR EW	./MCO

This directory contains all header files required for using the CANopen libraries. In order to allow easy future updates/upgrades and to ensure that the code remains CANopen conformant, these files should not be modified by the end user.

4.1.2 Library Directory

Environment	Path
Native/file system	./Example/mcolibs
MCUXpresso	./mcolibs
Keil uVision	n/a, library added to linker options
IAR EW	n/a, library added to linker options

This directory contains the CANopen library available for the used application, target architecture and protocol.

4.1.3 Configuration Directory

Environment	Path
Native/file system	./Example-Config
MCUXpresso	./mco_config
Keil uVision	./MCO-Config
IAR EW	./MCO-Config

This directory contains the files and modules configuring the CANopen device implemented. These files need to be modified or generated for each application.

File / Module	Content
procimg.h	Definition of process image access macros
nodecfg.h	CANopen functionality configuration
user_cbdata.c	Application call-back functions
user_od.c	Tables with Object Dictionary (pulls-in auto-generated entries from CANopen Architect)

4.1.3.1 EDS and EDS-FD Subdirectories:

Environment	Path
Native/file system	./Example-Config/EDS, ./Example-Config/EDS-FD
MCUXpresso	./mco_config/EDS, ./mco_config/EDS-FD
Keil uVision	./MCO-Config/EDS, ./MCO-Config/EDS-FD
IAR EW	./MCO-Config/EDS, ./MCO-Config/EDS-FD

This directory contains the application's EDS or XDD files (Electronic Data Sheet or Extensible Device Description File) as well as auto-generated source code files generated by the CANopen Editor "CANopen Architect". The auto-generated files should not be modified as any recreation of the files by CANopen Architect would overwrite any local modifications.

File / Module	Content
Application.caxe/.cax	CANopen Architect Mini and CANopen Architect Std/Pro project files
Application.eds/.xdd	Application's Electronic Data Sheet
Application.docx/pdf	Auto-generated documentation generated by CANopen Architect Pro
entriesandreplies.h stackinit.h pimg.h	Auto-generated configuration files generated by CANopen Architect

4.1.4 Application Directory

Environment	Path
Native/file system	./Example-Source
MCUXpresso	./source
Keil uVision	./source
IAR EW	./source

This directory contains the files and modules implementing the CANopen application. These files need to be modified or generated for each application.

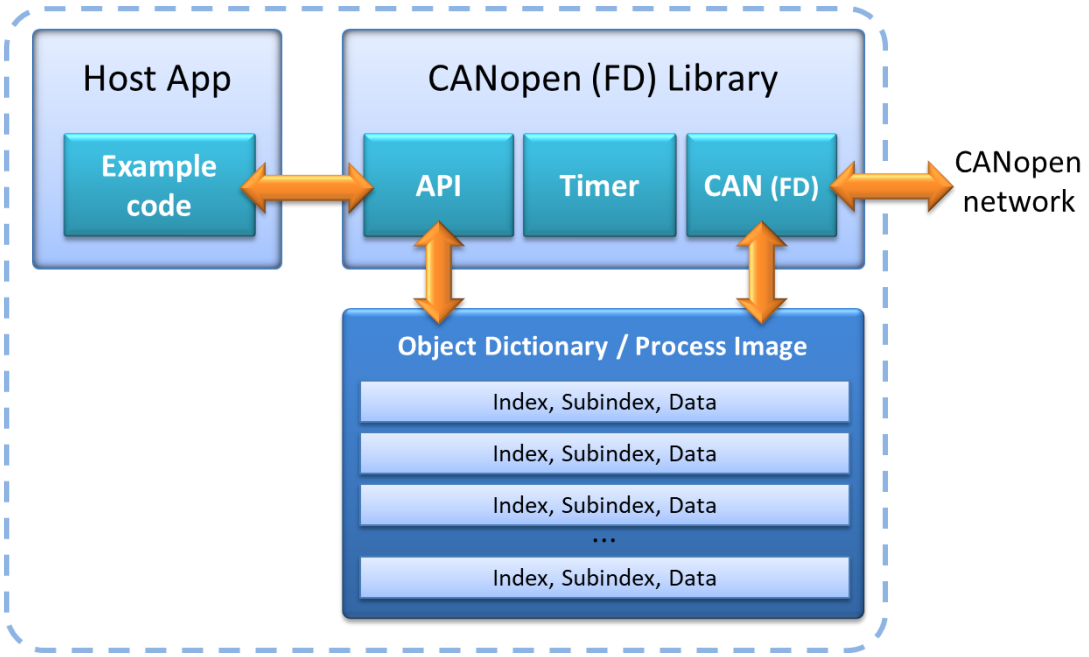
4.1.5 Example Project Directory

Environment	Path
Native/file system	./Example/[MCU_used]
MCUXpresso	./
Keil uVision	./source
IAR EW	./source

This directory contains the project files and settings for an example project, depending on MCU and target board used.

4.2 CANopen Functional Overview

The CANopen libraries can be used to implement CANopen and CANopen FD Slave nodes in accordance with almost any Device or Application profile available today.



The figure above illustrates the operation blocks of the CANopen examples. The object dictionary and process image layout is configured using the CANopen Architect utility, which generates the .h configuration files needed by application example and library. The example application has access to the process image through the API of the CANopen library which also has direct access to it when transmitting or receiving data on the CANopen network side.

4.2.1 Node ID

Every CANopen device on a CANopen network must have a unique node ID number. This is either assigned directly with the auto-generated configuration or it must be passed as a parameter when initializing the CANopen library.

The commercially available extended libraries support LSS (Layer Setting Services) where devices without a node ID can be detected through the CANopen network. An LSS Master can then assign these unconfigured devices a node ID to use.

4.2.2 Process Image Usage

All data communicated via CANopen (FD) is organized in a process image, an array of bytes. Data is referred to by an offset into the process image. These offsets can be auto-generated by the CANopen configuration tool CANopen Architect (file pimg.h).

The application program may access parameters in the process image at any time using the macros *PI_READ()* and *PI_WRITE()*.

4.2.3 Object Dictionary and SDO/USDO Server

The CANopen libraries implement an object dictionary with one or multiple SDO/USDO servers. Depending on the size of an OD entry, expedited or segmented SDO/USDO transfers are used. Expedited transfers are used for a size of up to 4 (CANopen) resp. 54 (CANopen FD) bytes. Otherwise segmented transfers are used.

Using the SDO/USDO server, SDO/USDO clients can send read/write requests to the Object Dictionary.

4.2.4 Heartbeat Producer

The CANopen libraries implement the heartbeat producer method.

As recommended by the CiA and other CANopen experts, the CANopen libraries implement the newer heartbeat method instead of the older node guarding method.

4.2.5 PDO Parameters

With these CANopen libraries, PDO mapping parameters (data mapped into the PDO) are static (hard-coded by configuration files created with CANopen Architect). The PDO communication parameters are dynamic (can be changed during operation). PDO communication parameters include CAN ID used and trigger options like change-of-state with an inhibit time, event timer (periodical) and SYNC.

4.2.6 Number of PDOs

The maximum number of PDOs supported by this library are 4 TPDOs and 4 RPDOs (16 TPDOS and 16 RPDOS for Manager). Extended versions of the library supporting all 512 PDOs are commercially available, see www.em-sa.com/nxp.

4.2.7 Emergency Producer and Emergency messages

The CANopen libraries support the production of emergency messages. Emergencies can be triggered by the application as well as by the CANopen stack, for example if a PDO received has a different length than expected.

Upon startup (right after boot up message) an Emergency clear message (code 0000h) is transmitted. Further, the default implementation of the user function MCOUSER_FatalError() generates an Emergency with the error code in the custom area of the Emergency message.

If an illegal NMT command (CAN message ID zero) is received, an Emergency with code 8200h is produced. The custom error area shows the illegal command byte received.

If a PDO received has a different length as expected, an Emergency with code 8210h is produced. The custom error area included the PDO number (lo byte and hi byte), the expected length and the length of the received PDO.

If Heartbeat consumption is enabled and a loss of a heartbeat is detected, then an Emergency with code 8130h is produced. The custom area shows the node ID number of the node whose heartbeat was lost.

4.2.8 Emergency Consumer

Only available in the Manager version of the library. The call-back function *MGRCB_NodeStatusChanged()* informs the application about emergencies received.

4.2.9 Heartbeat Consumer

This release of the CANopen libraries provide three heartbeat consumer channels for devices and 32 for Managers. The application is informed once a heartbeat monitored is lost. The channels can be configured both through the CANopen network as well as by the application.

4.2.10 Store Parameters

The CANopen library implements the store parameters functionality. This means that the current configuration of the CANopen device can be saved to non-volatile memory and will automatically be used after power-up.

Requires adaptation of the *nvol_xxxx.c* driver to store data in non-volatile memory.

4.2.11 Layer Setting Services

Not available in this release of the CANopen library.

5 Application Interface

Both shared data memory and function calls are used to implement an interface between CANopen and the application program. A process image (array of bytes) is used as shared memory that can be accessed from both the CANopen library as well as from the application program. The process image contains all process data variables that are communicated via CANopen. Access functions are provided to allow the application program to read or write data from or to the process image.

5.1 The Process Image

In order to offer a generic method for addressing and exchanging the data communicated via CANopen, the data is organized into a process image which is implemented as an array of bytes. The length of the process image in bytes is defined by *PROCIMG_SIZE* in file *procimg.h* and must be in the range of 1 to FFFEh (values 0 and FFFFh are reserved).

A single variable of the process image can be addressed by specifying an offset and a length. The offset specifies where in the process image the first byte of a variable is stored and the length specifies how many bytes are used to store the variable. The offset may have a value from 0 to FFFEh. Using an offset of FFFFh indicates that the offset is invalid or unused.

If numeric values are stored in multiple byte variables, then the default byte order is CANopen compatible: Little Endian – the lower bytes are stored at the lower offset.

5.1.1 Configuration of the Process Image

The process image configuration is automatically generated by CANopen Architect. The default file name for the file containing the process image variable definitions generated by CANopen Architect is *pimg.h*.

5.1.2 Accessing the Process Image

Only use the provided macros to access data in the process image. See chapter 5.3.7 Process Image Access Macros: The PI_READ macro and below for details.

5.1.3 Data alignment

In general, CANopen Architect can be configured to align the data in the process image. This ensures that 32bit values are stored at an address dividable by 4.

Unfortunately, that is not always possible. PDO mapped data is stored as one block. If data is unaligned in a PDO definition (as used in some CiA documents), then it will be unaligned in the process image, too. The definition of the *PI_READ()* and *PI_WRITE()* macros must be such, that also unaligned data is read properly.

5.1.4 Data Integrity of the Process Image in an RTOS Environment

The process image is accessed by both the application and the CANopen library (both with SDO/USDO and PDO accesses). Typically, locks are required to ensure data integrity.

To ease the implementation of such locks, all process image accesses need to be made using the macros PI_READ(), PI_WRITE() and PI_COMP(). The read and write macros need to be enhanced with custom code to create and release a lock before and after accessing the process image. These are defined in file procimg.h.

Note: PI_COMP() also executes an read access, however it is only used to detect a data change and therefore does not need to be protected.

5.2 Object Dictionary Configuration

The Object Dictionary configuration is automatically generated by CANopen Architect. The default file name for the file containing the process image variable definitions is *entriesandreplies.h*.

The file in which the OD is created is called *user_od.c* (User Object Dictionary file).

5.3 CANopen API Functions and Macros

This section lists all the functions that can be called by the application program.

5.3.1 The MCO_Init function

The MCO_Init function (re-)initializes the CANopen protocol stack. It needs to be called during system initialization. It may also be called to re-initialize the CANopen stack, for example to force a reset of the CANopen communication task(s).

Declaration

```
void MCO_Init (
  UNSIGNED16 Baudrate,    // CAN FD arbitration bitrate in kbps
  UNSIGNED16 BRSBaudRate,// CAN FD data bitrate in kbps, FD only
  UNSIGNED8 Node_ID,      // CANopen node ID (1-127)
   UNSIGNED16 Heartbeat // Heartbeat time in ms
   );
```

Passed

`Baudrate` selects the desired CAN bit rate to be used. The following values are typically used for CANopen:

1	use default or predefined bit rate
10	use 10 kbps
20	use 20 kbps
50	use 50 kbps
125	use 125 kbps
250	use 250 kbps
500	use 500 kbps
800	use 800 kbps
1000	use 1,000 kbps

`Node_ID` is the CANopen node ID to be used by this CANopen node. The allowed value range is 0 to 127. If 0 is selected, Micro CANopen Plus will use the default or preconfigured node ID.

`Heartbeat` is the heartbeat producer time in milliseconds. If set to zero, Micro CANopen Plus will try to use a default or predefined value.

Returned

Nothing.

5.3.2 The MCO_InitxPDOx functions

These functions initialize the PDOs used. PDO initialization is part of the auto-generated code from CANopen Architect in file *stackinit.h*.

5.3.3 The MCO_ProcessStack function

This function must be called periodically to keep the CANopen stack operating. With each call it is checked if the CAN receive queue contains a message that needs to be processed. Depending on configuration it is also checked if timers expired or process data changed. This is typically called from the main while(1) loop. For best operation this should be called at least once per millisecond. If called less often multiple calls should be executed (see return value below).

Declaration

```
UNSIGNED8 MCO_ProcessStack (
  void
  );
```

Passed

Nothing.

Returned

The return value is TRUE, if something was processed and FALSE if there was nothing to do. If called less frequent, like every few milliseconds this function should be called repeatedly until the return value is FALSE.

Example

```
  while (MCO_ProcessStack() == TRUE);
```

5.3.4 The MCO_TriggerTPDO function

This function may be called by the application when a TPDO should be transmitted. Can be called after a write to the process image to avoid lengthy auto-detection of a COS (Change Of State).

Declaration

```
void MCO_TriggerTPDO (
  UNSIGNED16 TPDONr  // TPDO number to transmit
  );
```

Passed

The parameter `TPDONr` defines the TPDO number to be triggered. Must be in range from 1 to NR_OF_TPDOS

Returned

Nothing.

5.3.5 The MCOP_InitHBConsumer function

When heartbeat consumer functionality is enabled, this function can be used to manually re-initialize a heartbeat consumer.

NOTE that under regular CANopen configuration the heartbeat consumers are initialized through configuration – setting the heartbeat consumer times using a CANopen configuration tool.

Declaration

```
void MCOP_InitHBConsumer (
  UNSIGNED8 consumer_channel, // HB Consumer channel
  UNSIGNED8 node_id, // Node ID to monitor
  UNSIGNED16 hb_time // Timeout to use (in ms)
  );
```

Passed

`consumer_channel` is the number of the heartbeat consumer channel that gets initialized with this call.

`node_id` is the CANopen node ID of the node monitored.

`hb_time` is the heartbeat timeout used in milliseconds. As a rule over thumb this should be a multiple of what the heartbeat producer timer is.

Returned

Nothing.

5.3.6 The MCOP_PushEMCY and MCOP_PushEMCYFD functions

When Emergency usage is enabled (*#define USE_EMCY*) functionality is enabled, a CANopen Emergency message can be transmitted with these function calls. The legacy PushEMCY can still be used, it calls the PushEMCYFD version with default parameters.

Declaration

```
UNSIGNED8 MCOP_PushEMCYFD
  (
  UNSIGNED8  logic,// logical device number (0 for unknown)
  UNSIGNED16 cia_doc, // 16 bit CiA spec number referred
  UNSIGNED16 emcy_code, // 16 bit error code
  UNSIGNED8  em_1, // 5 byte manufacturer
  UNSIGNED8  em_2, // specific error code
  UNSIGNED8  em_3,
  UNSIGNED8  em_4,
  UNSIGNED8  em_5,
  UNSIGNED8  status, // error priority, class, state
  TIME_OF_DAY time // time stamp, if available
  );
```

Passed

The logical device number causing the emergency.

The CiA document number defining the emergency code.

The 16-bit emergency error code as specified by the CANopen standards.

Up to five bytes of manufacturer specific emergency / error information.

Emergency status information as defined in CiA 1301.

Timestamp, if available.

Returned

True if the message was successfully added to the transmit queue.

Declaration

```
UNSIGNED8 MCOP_PushEMCY
  (
  UNSIGNED16 emcy_code, // 16 bit error code
  UNSIGNED8  em_1, // 5 byte manufacturer
  UNSIGNED8  em_2, // specific error code
  UNSIGNED8  em_3,
  UNSIGNED8  em_4,
  UNSIGNED8  em_5
  );
```

Passed

The 16-bit emergency error code as specified by the CANopen standards.

Up to five bytes of manufacturer specific emergency / error information.

Returned

True if the message was successfully added to the transmit queue.

5.3.7 Process Image Access Macros: The PI_READ macro

This macro is defined in procimg.h and used to execute read accesses from the process image. This can be customized or provided as a function if the application wants to have a direct call-back for any read access made from the process image.

Declaration

```
PI_READ(level,offset,pdst,len)
```

Passed

`level` indicates a priority level for the access, is set to PIACC_APP, PIACC_PDO or PIACC_SDO depending on if the access is made from the application, PDO processing or SDO/USDO processing.

`offset` is the offset into the process image to the location from which the read is executed.

`pdst` is a memory pointer to the destination to which data is copied.

`len` is the length of the data to be copied in bytes.

Returned

Nothing or length of data copied.

5.3.8 Process Image Access Macros: The PI_WRITE macro

This macro is defined in procimg.h and used to execute write accesses to the process image. This can be customized or provided as a function if the application wants to have a direct call-back for any write access made to the process image.

Declaration

```
PI_WRITE(level,offset,psrc,len)
```

Passed

`level` indicates a priority level for the access, is set to PIACC_APP, PIACC_PDO or PIACC_SDO depending on if the access is made from the application, PDO processing or SDO/USDO processing.

`offset` is the offset into the process image to the location to which the write is executed.

`psrc` is a memory pointer to the source from which data is copied.

`len` is the length of the data to be copied in bytes.

Returned

Nothing or length of data copied.

5.3.9 Process Image Access Macros: The PI_COMP macro

This macro is defined in procimg.h and used to compare data with data in the process image. This can be customized or provided as a function if the application wants to have a direct call-back for any compare access made to the process image.

Declaration

```
PI_COMP(level,offset,psrc,len)
```

Passed

`level` indicates a priority level for the access, is set to PIACC_APP, PIACC_PDO or PIACC_SDO depending on if the access is made from the application, PDO processing or SDO/USDO processing.

`offset` is the offset into the process image to the location that is to be compared.

`psrc` is a memory pointer to the data that is to be compared.

`len` is the length of the data to be compared in bytes.

Returned

0 if the data is identical and unequal 0 otherwise.

5.3.10 Default Process Image Access Macros

The code is delivered with default macros that use plain calls to memcpy resp. memcmp from the ANSI-C string library:

```
#define PI_READ(level,offset,pdst,len)    memcpy(pdst,&(gProcImg[offset]),len)
#define PI_WRITE(level,offset,psrc,len)   memcpy(&(gProcImg[offset]),psrc,len)
#define PI_COMP(level,offset,psrc,len)    memcmp(&(gProcImg[offset]),psrc,len)
```

In environments where the following is true, these will work fine:

- No RTOS is used
- The process image is not accessed from within interrupt service routines

5.3.11 Macros for PDO process image access

For all PDO-related accesses, Micro CANopen Plus uses dedicated macros:

`PDO_TXCOPY(TPDO,dat)`
Copy from process image to TPDO CAN message buffer

`PDO_RXCOPY(TPDO,dat)`
Copy from RPDO CAN message buffer to process image

`PDO_TXCOMP(TPDO,dat)`
Compare TPDO CAN message buffer with what is in the process image (for change-of-state detection)

These macros are defined using PI_READ, PI_WRITE and PI_COMP general access macros with PIACC_PDO as the first parameter for the access level.

5.4 CANopen API Call-Back Functions

This section lists the call-back functions that can be called by the CANopen protocol stack. They indicate important CANopen events to the application.

5.4.1 The MCOUSER_ResetCommunication function

This function is called to completely re-initialize the CANopen communication. This includes re-initialization of the CAN interface. This function is called upon initialization but also when the CANopen node received the NMT Master command to soft-reset itself.

Declaration

```
void MCOUSER_ResetCommunication (
  void
  );
```

Passed

Nothing.

Returned

Nothing.

5.4.2 The MCOUSER_ResetApplication function

This function is called when the CANopen node received the command from the NMT Master to hard-reset itself. Both the CANopen communication as well as the application is expected to fully reset. This is typically implemented using a watchdog reset.

Declaration

```
void MCOUSER_ResetApplication (
  void
  );
```

Passed

Nothing.

Returned

Nothing.

5.4.3 The MCOUSER_GetSerial function

This function is called before read accesses to the Object Dictionary entry [1018h,0] – Serial Number. It can be used by the application to retrieve the serial number, for example from non-volatile memory.

Declaration

```
UNSIGNED32 MCOUSER_GetSerial (
  void
  );
```

Passed

Nothing.

Returned

The 32-bit serial number of the device.

5.4.4 The MCOUSER_NMTChange function

This Micro CANopen Plus function only exists if the compiler directive USECB_NMTCHANGE is defined. It is then called whenever the CANopen protocol stack changes the NMT Slave state – typically this happens after receiving the NMT Master Message.

Declaration

```
void MCOUSER_NMTChange (
  UNSIGNED8 NMTState
  );
```

Passed

The value for NMTSTATE indicates the current NMT Slave State. It can be one of the following values: NMTSTATE_BOOT (0), NMTSTATE_STOP (4), NMTSTATE_OP (5) or NMTSTATE_PREOP (127).

00h Initializing (sent after receiving the 'I' command)
04h CANopen NMT state "stopped" entered
05h CANopen NMT state "operational" entered
7Fh CANopen NMT state "pre-operational" entered

Returned

Nothing.

5.4.5 The MCOUSER_FatalError function

This indication signals the application that the CANopen stack ran into a fatal error situation and needs to be reset or re-initialized to start operation again.

Declaration

```
void MCOUSER_FatalError (
  UNSIGNED16 ErrCode  // the error code
  );
```

Passed

The `ErrCode` is an internal 16-bit error code. As a general rule, error codes below 8000h indicate a warning and the stack CANopen could still continue operation. However, an error code of 8000h or higher indicates a fatal error requiring re-initialization or a reset of the system.

Returned

Nothing.

5.4.6 The MCOUSER_ODData function

This Micro CANopen Plus function is only available when the compiler directive USECB_ODDATARECEIVED is set to one. The function signals the receipt of process data stored into the process image, no matter if it came in by PDO or SDO/USDO transfer.

Declaration

```
void MCOUSER_ODData (
  UNSIGNED8 client_nid,      // node ID from where data arrived (0 if unknown)
  UNSIGNED16 idx,            // Index
  UNSIGNED8 subidx,          // Subindex
  UNSIGNED8 MEM_PROC *pDat,  // pointer to data
  UNSIGNED16 len             // length of data
  );
```

Passed

The parameters passed include the requesting client's node ID, if known (only if a USDO access triggered this), Index and Subindex of the data received into the Object Dictionary as well as a pointer to the data and the length of the data in bytes.

Returned

Nothing.

5.4.7 The MCOUSER_SYNCReceived function

This Micro CANopen Plus function is only available when the compiler directive USECB_SYNCRECEIVE is defined. The function signals the receipt of the CANopen SYNC message. Synchronous TPDO data transmission will be triggered and synchronous RPDO will be received after execution of this call-back function.

Declaration

```
void MCOUSER_SYNCReceived (
  void
  );
```

Passed

Nothing.

Returned

Nothing.

5.5 Manager API

This section lists all the functions that can be called by the application program.

5.5.1 The MGR_ProcessMgr function

This function must be called periodically to keep the Manager operating. With each call it is checked if the CAN receive queue contains a message for the Manager that needs to be processed. Depending on configuration it is also checked if timers expired (like USDO response timeouts). This is typically called from the main while(1) loop. If called less often multiple calls should be executed (see return value below).

Declaration

```
UNSIGNED8 MGR_ProcessMgr (
  void
  );
```

Passed

Nothing.

Returned

The return value is TRUE, if something was processed and FALSE if there was nothing to do. If called less frequent, like every few milliseconds this function should be called repeatedly until the return value is FALSE.

```
  while (MGR_ProcessMgr() == TRUE);
```

5.5.2 The MGR_TransmitNMT function

This function may be called by the application to transmit the Network Management Master Message.

Declaration

```
UNSIGNED8 MGR_TransmitNMT (
  UNSIGNED8 nmt_cmd, // NMT command
  UNSIGNED8 node_id  // Node ID, or zero for all nodes
  );
```

Passed

The parameter `nmt_cmd` defines the NMT command transmitted. The following commands are available:

NMTMSG_PREOP	Pre-operational
NMTMSG_OP	Operational
NMTMSG_STOP	Stop
NMTMSG_RESETCOM	Reset Communication
NMTMSG_RESETAPP	Reset Application

Returned

True, if the message was successfully added to the CAN transmit queue.

5.6 Manager Multi-Scan API

The manager can use an SDO or USDO client to perform multiple sequential reads from a node and collect the results in a table. The call-back MGRCB_NodeStatusChanged informs the application when scanning is completed.

5.6.1 The MGRSCAN_Init function

This function initializes the auto scan feature of the manager.

Declaration

```
void MGRSCAN_Init (
  UNSIGNED8 sdo_clnt,   // SDO client number to use
  UNSIGNED8 node_id,    // Node ID of node to read data from
  UNSIGNED8 *pScanList, // list with OD entries to be read
  UNSIGNED32 *pScanData,// Pointer to destination array
  UNSIGNED16 Delay      // Delay in ms between read requests
  );
```

Passed

The parameter `sdo_clnt` defines the (U)SDO channel number used, this must be in between one and NR_OF_SDO_CLIENTS.

The node to read from is selected by `node_id`.

The parameter `pScanList` must point to a table with 32bit entries:

Index(16bit), Subindex(8bit), Length(8bit)

Last entry must be 0xFFFFFFFF

The destination buffer `pScanData` must be big enough to hold all entries back to back.

The parameter `Delay` specifies a timeout in milliseconds used between transfers to avoid producing too much of a busload.

Returned

Nothing.

5.6.2 The MGRSCAN_GetStatus function

This function can be used to quickly check if the auto scan is still running or not.

Declaration

```
UNSIGNED8 MGRSCAN_GetStatus (
  UNSIGNED8 node_id     // Node ID of node which is scanned
);
```

Passed

The node id of the node which is currently scanned.

Returned

TRUE if scan is still in progress, else FALSE.

5.7 Manager API Call-Back Functions

This section lists all call-back functions that can be called by the CANopen Manager. They indicate events to the application.

5.7.1 The MGRCB_NodeStatusChanged function

This function is called when the status of any of the nodes on the network has changed. The changes reported are:

NODESTATUS_BOOT	Node booted
NODESTATUS_PREOPERATIONAL	Node changed into pre-operational
NODESTATUS_OPERATIONAL	Node changed into operational mode
NODESTATUS_STOPPED	Node changed into stopped mode
NODESTATUS_HBACTIVE	Heartbeatmonitoring is now active
NODESTATUS_HBLOST	Heartbeat was lost
NODESTATUS_SCANCOMPLETE	Auto-scan of node completed
NODESTATUS_EMCY_OVER	Emergency clear
NODESTATUS_EMCY_NEW	New emergency occurred

For the last three more detailed information can be found in the nodelist structure.

Declaration

```
void MGRCB_NodeStatusChanged (
  UNSIGNED8 node_id,
  UNSIGNED8 status
  );
```

Passed

The node ID number of the node whose status changed and the status change reported.

Returned

Nothing.

5.8 SDO Client API

SDO client services allow CANopen devices to actively send SDO requests to other devices. This allows access to the Object Dictionary of all nodes connected to a network. Functions provided allow reading or writing an entire list of OD entries. SDO client services may in general also be used by CANopen slave devices. However, most common usage is in Managers.

5.8.1 The SDOCLNT_ResetChannels function

This function resets all SDO Client channels

Declaration

```
void SDOCLNT_ResetChannels (
  );
```

Passed

Nothing.

Returns

Nothing.

5.8.2 The SDOCLNT_Init function

This function initializes an SDO client as needed for sending SDO requests to CANopen devices.

Declaration

```
SDOCLIENT *SDOCLNT_InitSDOClient (
  UNSIGNED8 channel, // SDO channel number
  UNSIGNED32 canid_request, // CAN message ID SDO request
  UNSIGNED32 canid_response, // CAN message ID SDO response
  UNSIGNED8 *p_buf, // data buffer for data exchanged
  UNSIGNED32 buf_size // max length of data buffer
  );
```

Passed

The parameter `channel` defines the SDO channel number used, the number of channels available is determined by NR_OF_SDO_CLIENTS.

The CAN IDs used for the transmitted request and the expected response are determined by the parameters `canid_request` and `canid_response`.

The parameters `p_buf` and `buf_size` define the data buffer used to hold data transmitted or received.

Returned

This function returns a pointer to data structure of type `SDOCLIENT`. This data structure holds all information for this SDO channel and the pointer is needed for referencing this SDO channel when using the read and write functions.

If the pointer returned is zero, then initialization failed.

5.8.3 The SDOCLNT_Read function

This function starts an SDO read process. The function is non-blocking and returns before the response is received. Once a response is received or a timeout occurs, the call back function *MGRCB_SDOComplete* is called. The data received will be made available in the buffer specified during initialization of the client.

Declaration

```
UNSIGNED8 SDOCLNT_SDOClientRead (
  SDOCLIENT *p_client, // Pointer to SDO client structure
  UNSIGNED16 index, // Object Dictionary Index to read
```

```
  UNSIGNED8 subindex // Object Dictionary Subindex to read
  );
```

Passed

The parameter `p_client` is a pointer to the SDO client structure returned during initialization..

The parameters `index` and `subindex` define the object dictionary entry to be read.

Returned

True, if the client request was queued to the CAN transmit queue.

5.8.4 The SDOCLNT_ReadXtd function

This function starts an SDO read process with extended parameters. In addition to SDOCLNT_READ() a receive buffer and an individual timeout can be specified.

Declaration

```
UNSIGNED8 SDOCLNT_ReadXtd (
  SDOCLIENT *p_client, // Pointer to initialized SDO client
  UNSIGNED16 index, // Object Dictionary Index to read
  UNSIGNED8 subindex, // Object Dictionary Subindex to read
  UNSIGNED8 *pDest, // Pointer to data destination
  UNSIGNED32 len, // Maximum length of data destination
  UNSIGNED16 timeout // Timeout for this transfer in ms
  );
```

Passed

The parameter `p_client` is a pointer to the SDO client structure returned during initialization..

The parameters `index` and `subindex` define the object dictionary entry to be read.

The parameters `pDest` and `len` define the destination buffer for the data read.

The parameter `timeout` specifies the timeout in milliseconds.

Returned

True, if the client request was queued to the CAN transmit queue.

5.8.5 The SDOCLNT_Write function

This function starts an SDO write process. The function is non-blocking and returns before the response is received. Once a response is received or a timeout occurs, the call back function *MGRCB_SDOComplete* is called. The data transmitted must already be present in the buffer specified during initialization of the client.

Declaration

```
UNSIGNED8 MGR_SDOClientWrite (
  SDOCLIENT *p_client, // Pointer to SDO client structure
  UNSIGNED16 index, // Object Dictionary Index to write
  UNSIGNED8 subindex // Object Dictionary Subindex to write
  );
```

Passed

The parameter `p_client` is a pointer to the SDO client structure returned during initialization..

The parameters `index` and `subindex` define the object dictionary entry to be written to.

Returned

True, if the client request was queued to the CAN transmit queue.

5.8.6 The SDOCLNT_WriteXtd function

This function starts an SDO write process with extended parameters. In additon to the regular call a source buffer and a timeout can be specified.

Declaration

```
UNSIGNED8 SDOCLNT_WriteXtd (
  SDOCLIENT *p_client, // Pointer to initialized SDO client
  UNSIGNED16 index, // Object Dictionary Index to write
  UNSIGNED8 subindex, // Object Dictionary Subindex to write
  UNSIGNED8 *pSrc, // Pointer to data source
  UNSIGNED32 len, // Length of data
  UNSIGNED16 timeout // Timeout for this transfer in ms
  );
```

Passed

The parameter `p_client` is a pointer to the SDO client structure returned during initialization..

The parameters `index` and `subindex` define the object dictionary entry to be written to.

The parameters `pSrc` and `len` define the source buffer for the data write.

The parameter `timeout` specifies the timeout in milliseconds.

Returned

True, if the client request was queued to the CAN transmit queue.

5.8.7 The SDOCLNT_GetStatus function

Returns the current status of an SDO client.

Declaration

```
UNSIGNED32 SDOCLNT_GetStatus (
  SDOCLIENT *p_client // Pointer to initialized SDO client
  );
```

Passed

The parameter `p_client` is a pointer to an SDOCLIENT structure.

Returned

The status information for this SDO client.

SDOERR_FATAL	Illegal pointer passed
SDOERR_OK	Confirmation, last access was success
SDOERR_ABORT	Abort received
SDOERR_TIMEOUT	Request timed out
SDOERR_TOGGLE	Toggle error
SDOERR_BUFSIZE	Out of Memory
SDOERR_UNKNOWN	No transfer possible
SDOERR_RUNNING	SDO Transfer still running, not complete

5.8.8 The SDOCLNT_GetLastAbort function

Returns the last abort code of a SDO client.

Declaration

```
UNSIGNED32 SDOCLNT_GetLastAbort (
  SDOCLIENT *p_client // Pointer to initialized SDO client
  );
```

Passed

The parameter `p_client` is a pointer to an SDOCLIENT structure.

Returns

The last SDO abort code. A value of 0xFFFFFFFF or zero indicates that no abort code is available.

5.8.9 The SDOCLNT_BlockUntilCompleted function

Waits until a SDO client transfer completed (received response, abort or timeout).

Declaration

```
UNSIGNED32 SDOCLNT_BlockUntilCompleted (
  SDOCLIENT *p_client // Pointer to initialized SDO client
  );
```

Passed

The parameter `p_client` is a pointer to an SDOCLIENT structure.

Returned

SDO_ERR status code, same as SDOCLNT_GetStatus.

5.8.10 The SDOCLNT_ReadCycle function

This function executes an entire SDO read cycle including initialization of an SDO channel, sending the SDO request and waiting for the response. As the function is blocking (waits until cycle completed) it should only be used when an RTOS is used. During the wait loop the macro RTOS_SLEEP is used, this needs to be defined to either a fixed timeout (like 1 to 5 milliseconds) or to a “wait until” the next CAN message was received and placed into the manager’s CAN receive queue.

Declaration

```
UNSIGNED32 MGR_SDOClientReadCycle (
  UNSIGNED8 channel, // SDO chn number 1-NR_OF_SDO_CLIENTS
  UNSIGNED8 node_id, // CANopen node ID
  UNSIGNED16 index, // Object Dictionary Index to read
  UNSIGNED8 subindex, // Object Dictionary Subindex to read
  UNSIGNED8 *p_buf, // data buffer pointer for data rec.
  UNSIGNED32 buf_size // max length of data buffer
  );
```

Passed

The parameter `channel` is the SDO channel number used for this transfer.

With the parameter `node_id` the CANopen slave node is selected to which the SDO read request is send.

The parameters `index` and `subindex` define the object dictionary entry to be read.

The pointer `p_buf` and the size parameter `buf_size` define the data buffer into which the received data is copied.

Returned

On success returns the number of bytes of received data, else zero.

5.8.11 The SDOCLNT_WriteCycle function

This function executes an entire SDO write cycle including initialization of an SDO channel, sending the SDO request and waiting for the response. As the function is blocking (waits until cycle completed) it should only be used when an RTOS is used. During the wait loop the macro RTOS_SLEEP is used, this needs to be defined to either a fixed timeout (like 1 to 5 milliseconds) or to a “wait until” the next CAN message was received and placed into the manager’s CAN receive queue.

Declaration

```
UNSIGNED32 MGR_SDOClientWriteCycle (
  UNSIGNED8 channel, // SDO chn number 1-NR_OF_SDO_CLIENTS
  UNSIGNED8 node_id, // CANopen node ID
  UNSIGNED16 index, // Object Dictionary Index to read
  UNSIGNED8 subindex, // Object Dictionary Subindex to read
  UNSIGNED8 *p_buf, // data buffer pointer for data rec.
  UNSIGNED32 buf_size // max length of data buffer
  );
```

Passed

The parameter `channel` is the SDO channel number used for this transfer.

With the parameter `node_id` the CANopen slave node is selected to which the SDO write request is send.

The parameters `index` and `subindex` define the object dictionary entry to be read.

The pointer `p_buf` and the size parameter `buf_size` define the data send to the specified Object Dictionary entry. All bytes in the data buffer are send.

Returned

On success returns the number of bytes of data transferred, else zero.

5.8.12 The SDOCLNTCB_SDOComplete call-back function

This function is called when an initiated SDO transfer was completed or aborted. If called to indicate the completion of a read request, then the data received is stored in the buffer specified during initialization of the SDO channel.

Declaration

```
void SDOCLNTCB_SDOComplete (
  UNSIGNED8 channel, // SDO channel number
  UNSIGNED32 abort_code // status, error, abort code
  );
```

Passed

The SDO `channel` number and the `abort_code`. The abort code is set to SDOERR_READOK or SDO-ERR_WRITEOK to indicate that the transfer completed without errors.

Returned

Nothing.

6 Appendix – Using Auto-Generated Sources

The CANopen EDS Editor "CANopen Architect" can generate source files directly usable by Micro CANopen Plus. This chapter summarizes the steps that need to be taken to generate the files and integrate them to Micro CANopen Plus based applications.

The application examples provided with Micro CANopen Plus have their EDS, DCF and auto-generated files stored in the directory MCO_APPLICATIONNAME/EDS/

6.1 File Generation

When editing an EDS or DCF with CANopen Architect some extra care should be taken when defining the access type for the Object Dictionary entry.

If the access type of an entry is CONST (constant), then CANopen Architect will not place the entry into the process image but will try to locate it in the non-volatile code space area. This helps to conserve the limited space available for process image data.

As an example, the entries [1008h-100Ah,00h] should be specified as CONST, as these are constant, read-only strings.

For entries using multiple subindexes, the first subindex entry (subindex 0) should also be marked as type CONST. CANopen Architect then places these into the SDO Reply table and not into the process image.

To generate the source files from CANopen Architect, simply select the menu "File | Export C Sources Files...". It is recommended to use the default file names suggested when exporting the files.

6.2 File Integration

This section describes the information found in each of the generated files and how these files need to be integrated into the application.

6.2.1 pimg.h

The file *pimg.h* contains the basic #define settings required by Micro CANopen Plus and all process image offset and size definitions for variables stored in the process image.

This file needs to be included to all the application's C source files that make accesses to data contained in the process image.

6.2.2 stackinit.h

The file *stackinit.h* contains auto-generated calls to the functions MCO_InitRPDO and MCO_InitTPDO which initialize the PDOs. The calls are provided as macro INITPDOS_CALLS.

The file also contains auto-generated calls to the functions MCO_InitHBConsumer to set up heartbeat consumer channels. The calls are provided as macro INITHBCONSUMER_CALLS.

This file should be included to the C source file initializing the CANopen stack and making the call to MCO_Init. This is typically the file *user_xxx.c* and the call to MCO_Init is made in MCOUSER_ResetCommunication.

The recommended use is:

```
if (MCO_Init(can_bps,node_id,DEFAULT_HEARTBEAT))
{
  //Initialization of PDOs comes from EDS
  INITPDOS_CALLS
  INITHBCONSUMER_CALLS
```

```
}
```

Note: If the CANopen Manager Add-on is used, the heartbeat consumers must be initialized later, after MGR_InitMgr() has been called to initialize the manager.

6.2.3 entriesandreplies.h

The file *entriesandreplies.h* contains all auto-generated Object Dictionary entries. These are provided as macros and can directly be included into the data tables defined in the *user_od.c* file.

Use Example:

```
...
#include "EDS/entriesandreplies.h"
...

// Table with SDO Responses for read requests to OD
UNSIGNED8 MEM_CONST gSDOResponseTable[] = {
  // Include file generated by CANopen Architect
  SDOREPLY_ENTRIES
  // End-of-table marker
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF
};

// Table with Object Dictionary entries to process Data
OD_PROCESS_DATA_ENTRY MEM_CONST gODProcTable[] =
{
  ODENTRY_ENTRIES
  // End-of-table marker
  ODENTRY(0xFFFF,0xFF,0xFF,0xFFFF)
};

#ifdef USE_EXTENDED_SDO
// Table with generic entries to memory
OD_GENERIC_DATA_ENTRY MEM_CONST gODGenericTable[] =
{
  ODGENTRY_ENTRIES
  ODGENTRYP(0xFFFF,0xFF,0xFF,0xFFFF,0xFFFFF)
};
#endif // USE_EXTENDED_SDO
```

7 Appendix – RTOS Integration

The most simplistic way to integrate CANopen with a Real-Time Operating System is to call *MCO_ProcessStack* periodically, for example in the main while(1) loop or from a one millisecond timer task.

7.1 RTOS Task: Receive and Tick

A more advanced configuration would not use *MCO_ProcessStack* at all, but the mayor two sub functions *MCO_ProcessStackRx* and *MCO_ProcessStackTick.*

In an RTOS environment, the driver function *MCOHW_PullMessage* should be implemented waiting / blocking and only return when a CAN message was received. The function *MCO_ProcessStackRx* can then be executed repeatedly without further delay in its own task.

```
for(;;) MCO_ProcessStackRx();
```

The function *MCO_ProcessStackTick* should be called with every RTOS timer tick. If a tick of 1ms or smaller is used, a single call is sufficient. If the RTOS tick is greater than one, then *MCO_ProcessStackTick* should be called repeatedly as long as the return value is TRUE.

```
while (MCO_ProcessStackTick() == TRUE);
```

7.2 Process Image Integrity

In order to protect the process image from multiple accesses “at the same time”, the tasks accessing it need to lock it as a single resource. To ease the implementation of such locks, all process image accesses (also from the application) must be made using the macros PI_READ(), PI_WRITE() and PI_COMP().

These macros need to be customized to implement a mutex or single token semaphore lock before making the access and a release /free of the mutex / semaphore after the access.

8 Appendix – CANopen Code Configuration

With the CANopen library the following settings can not be changed.
This appendix is informational only.

The file *nodecfg.h* contains the *#define* settings that configure and enable specific CANopen code functionality. The file settings in *procimg.h* specify the access to the process image. The settings in *mcohw.h* define hardware related settings.

Source code configuration files can automatically be generated by the CANopen Architect.

8.1 Default Configuration of nodecfg.h

8.1.1 #define ENFORCE_DEFAULT_CONFIGURATION [0|1]

This setting enables the default configuration. This is the only fully tested configuration. All other configuration options are provided for customer specific optimizations.

8.2 General Settings of nodecfg.h

8.2.1 #define USE_MCOP [1]

Legacy, must be set to 1.

8.2.2 #define CHECK_PARAMETERS [0|1]

If CHECK_PARAMETERS is enabled, additional code is generated that does plausibility checks upon entry of code functions, such as checking if parameters are within the allowed range. If a parameter is out of range, a call to *MCOUSER_FatalError()* is executed.

8.2.3 #define USE_LEDS [0|1]

Setting USE_LEDS to 1 enables two CANopen indicator lights as specified by the CiA document DR303. Both a RUN and ERR light are supported. When using this option, additional defines must be used for the physical switching of each light. These are LED_RUN_ON and LED_RUN_OFF for the RUN LED and LED_ERR_ON and LED_ERR_OFF for the ERR LED.

8.3 PDO Settings of nodecfg.h

8.3.1 #define NR_OF_RPDOS [num]

This value defines the number of RPDOs (Receive Process Data Objects) implemented. The value range is from 0 to 512.

8.3.2 #define NR_OF_TPDOS [num]

This value defines the number of TPDOs (Transmit Process Data Objects) implemented. The value range may be from 0 to 512.

8.3.3 #define USE_EVENT_TIME [0|1]

If *USE_EVENT_TIME* is enabled, TPDO trigger events may include using the event timer (periodic transmission every X milliseconds).

8.3.4 #define USE_INHIBIT_TIME [0|1]

If *USE_INHIBIT_TIME* is enabled, TPDO trigger events may include COS (Change Of State) detection with using the inhibit time.

NOTE:
Internally all inhibit times are calculated and used based on a resolution of one millisecond. However, CANopen specifies the inhibit time with a resolution of 100 microseconds. To be CANopen compatible, Micro CANopen Plus automatically does a divide or multiply by 10 when communicating the inhibit time via SDO/USDO requests/responses.

8.3.5 #define USE_SYNC [0|1]

If *USE_SYNC* is enabled, the PDOs support synchronized transmission. To activate SYNC transmission, a configuration tool needs to write the appropriate values to the transmission type field of the PDO communication parameters.

8.3.6 #define USE_DYNAMIC_PDO_MAPPING [0|1]

If the optional (available as order option) *USE_DYNAMIC_PDO_MAPPING* is enabled, the PDOs support dynamic mapping and multi-mapping. With dynamic mapping, the PDO mapping can be changed at runtime. This allows changing which Object Dictionary entries are transmitted/received in a PDO. In addition, multi-mapping is supported, which allows one Object Dictionary entry to be mapped to multiple PDOs.

8.4 NMT Service Settings of nodecfg.h

8.4.1 #define AUTOSTART [0|1]

When AUTOSTART is enabled, the CANopen device directly switches itself into the operational state after power-on or reset without waiting for a CANopen NMT Master message with an operational command.

8.4.2 #define DEFAULT_HEARTBEAT [ms]

The Object Dictionary entry [1017h,00h] Heartbeat Producer Time is implemented as read-write. The DEFAULT_HEARTBEAT defines the default heartbeat time used by Micro CANopen Plus and is specified in milliseconds.

8.4.3 #define DYNAMIC_HEARTBEAT_CONSUMER [0|1], #define NR_HB_CONSUMER [num]

When DYNAMIC_HEARTBEAT_CONSUMER is enabled, the Object Dictionary entries [1016h,xx] Heartbeat Consumer are implemented as read-write and can be changed through configuration. Otherwise they are hard-coded and cannot change during operation.

NR_HB_CONSUMER defines if the heartbeat consumer functionality is enabled. If this define is set to 0, the heartbeat consumer functionality is disabled. If unequal zero, it defines the maximum number of channels implemented, directly specifying the number of CANopen nodes that can be monitored.

8.4.4 #define USE_EMCY [0|1], #define ERROR_FIELD_SIZE [num]

When USE_EMCY is enabled, Micro CANopen Plus supports the generation of emergency messages. Emergencies are generated after each reset ("No Error" Emergency Message), upon critical failures (such as receiving a PDO with an illegal length) and upon application specific emergency events. Emergencies transmitted are copied into a error history, the predefined error field [1003h]. The size of the error history (in number of errors saved) is defined using ERROR_FIELD_SIZE.

See also chapter 4.2.7 Emergency Producer and Emergency messages for an overview of auto-generated emergency messages.

8.4.5 #define USE_NODE_GUARDING [0]

CANopen experts do not recommend the usage of node guarding. Instead, the newer heartbeat method should be used. However, to be compliant with legacy devices, Micro CANopen Plus supports minimal node guarding functionality that is enabled if this setting is enabled.

Must be zero for CANopen FD.

8.4.6 #define USE_STORE_PARAMETERS [0|1], #define NVOL_STORE_START [num], #define NVOL_STORE_SIZE [num]

When USE_STORE_PARAMETERS is enabled, the Store Parameters functionality of Micro CANopen Plus is available. The module `storpara.c` is required for this functionality.

When USE_STORE_PARAMETERS is enabled, the define NVOL_STORE_START must be set to the first usable address in the non-volatile memory. The default is zero. The application could use a value of greater than zero to reserve/protect a memory area in the non-volatile memory from accesses by the store parameters functionality. The functions of the store parameters module will not access non-volatile memory outside the window defined by NVOL_STORE_START and NVOL_STORE_SIZE. In case the window size is too small, the function MCOUSER_FatalError will be called.

8.4.7 #define NR_OF_SDOSERVER [0]

Defines the number of SDO servers implemented. A value of greater than one is currently only supported for CiA447 (car add-on devices) applications.

Set to zero for CANopen FD.

8.4.8 #define USE_SLEEP [0|1]

Defines if the sleep mode as first introduced by CiA447-1 is implemented. If enabled, the call-back function `MCOUSER_Sleep()` must be implemented.

8.5 CANopen FD Settings of nodecfg.h

These are the settings available for USDO servers. Note that USDO Client related settings are described in the Micro CANopen Manager manual for consistency.

8.5.1 #define USE_CANOPEN_FD [0|1]

Only effective with CANopen FD modules available in the project.

Enables CANopen FD related functionality in base stack and add ons. In particular, all uses of the SDO protocol are replaced by USDO.

8.5.2 NR_OF_USDO_CONNECTIONS [2]

For USDO servers, defines the number of USDO connections that the server can handle in parallel before it aborts a new connection attempt.

8.5.3 USDOSEGSRVRX_B2B_PROC [0]

For USDO servers, in USDO block downloads, this minimum processing time (in .1 milliseconds) for back-to-back segments is sent to the requesting client. Use 0 to disable.

8.5.4 USDOSEGSRVRX_REQ_TIMEOUT [2500]

A USDO segmented or block receive connection will only stay open if the next request arrives within this timeout (milliseconds).

8.6 Other Settings of nodecfg.h

8.6.1 #define USE_CiA447 [0]

Enables the CiA 447 specific support for the device profile for car add-on devices. This will need the CiA447 add-on module to Micro CANopen Plus to build.

Must be set to zero for CANopen FD.

8.6.2 #define USE_SDOMESH [0]

Enables the SDO fully-meshed setup for SDO communication in any direction between up to 16 nodes in a network.

Must be set to zero for CANopen FD.

8.7 User Call-Back Functions of nodecfg.h

8.7.1 #define USECB_NMTCHANGE [0|1]

When USECB_NMTCHANGE is enabled, Micro CANopen Plus uses the call-back function MCOUSER_NMTChange to signal a change in the NMT Slave State to the application.

8.7.2 #define USECB_SYNCRECEIVE [0|1]

When USECB_SYNCRECEIVE is enabled, Micro CANopen Plus uses the call-back function MCOUSER_SYNCReceived to signal the reception of the SYNC signal to the application.

8.7.3 #define USECB_RPDORECEIVE [0|1]

When USECB_RPDORECEIVE is enabled, Micro CANopen Plus uses the call-back function MCOUSER_RPDOReceived to signal the reception of an RPDO to the application.

8.7.4 #define USECB_ODDATARECEIVED [0|1]

When USECB_ODDATARECEIVED is enabled, Micro CANopen Plus uses the call-back function MCOUSER_ODData to signal the application that data was received and copied into the process image. This is called for both PDO and SDO/USDO accesses.

8.7.5 #define USECB_TPDORDY [0|1]

When USECB_TPDORDY is enabled, Micro CANopen Plus calls the function MCOUSER_TPDOReady right before it sends a TPDO. This allows the application to update the TPDO data before it is sent, if necessary.

8.7.6 #define USECB_SDOREQ [0|1]

When USECB_SDOREQ is enabled, Micro CANopen Plus uses the call-back function MCOUSER_SDORequest to signal the reception of an unknown SDO/USDO request to the application.

8.7.7 #define USECB_SDO_RD_PI [0|1]

When USECB_SDO_RD_PI is enabled, Micro CANopen Plus uses the call-back function MCOUSER_SDORdPI to signal to the application, that an SDO/USDO read request for data located in the process image was received. The call-back is executed BEFORE Micro CANopen Plus executes the read, allowing the application to either update the data or deny access to it.

8.7.8 #define USECB_SDO_RD_AFTER [0|1]

When USECB_SDO_RD_AFTER is enabled, Micro CANopen Plus uses the call-back function MCOUSER_SDORdAft to signal to the application, that an SDO/USDO read request for data located in the process image was executed. The call-back is executed AFTER Micro CANopen Plus executes the read, allowing the application to mark the data as read or clear it.

8.7.9 #define USECB_SDO_WR_PI [0|1]

When USECB_SDO_WR_PI is enabled, Micro CANopen Plus uses the call-back function MCOUSER_SDOWrPI to signal to the application, that an SDO/USDO write request for data stored in the process image was received. The call-back is executed BEFORE Micro CANopen Plus copies the data to the process image, allowing the application to verify the data (e.g. execute a range check).

8.7.10 #define USECB_SDO_WR_AFTER [0|1]

When USECB_SDO_WR_AFTER is enabled, Micro CANopen Plus uses the call-back function MCOUSER_SDOWrAft to signal to the application, that an SDO/USDO write request for data located in the process image was executed. The call-back is executed AFTER Micro CANopen Plus executes the write, allowing the application to now use the data received.

8.7.11 #define USECB_APPSDO_READ [0|1]

When USECB_APPSDO_READ is enabled, Micro CANopen Plus uses the call-back function MCOUSER_AppSDOReadInit to allow the application to implement access to readable, custom Object Dictionary entries of various lengths. One usage example would be a text buffer that can contain messages of different lengths.

The parameters for MCOUSER_AppSDOReadInit also include return values for a size and pointer – these can be used to inform Micro CANopen Plus of the location and size of the buffer that contains the "response".

8.7.12 #define USECB_APPSDO_WRITE [0|1]

When USECB_APPSDO_WRITE is enabled, Micro CANopen Plus uses the call-back functions MCOUSER_AppSDOWriteInit and MCOUSER_AppSDOWriteComplete to allow the application to implement access to writable, custom Object Dictionary entries of various lengths. One usage example would be text display that can accept text messages of various length.

The parameters for MCOUSER_AppSDOWriteInit also include return values for a receive buffer and its size. Micro CANopen Plus copies the data received to the location specified.

With MCOUSER_AppSDOWriteComplete Micro CANopen Plus informs the application that data was received and now has to be processed. The parameter "more" indicates if all data was received or more will follow, in which case the application needs to read all data from the buffer as it will be overwritten with the following data segments.

8.8 Manager and common SDO/USDO settings of nodecfg.h

8.8.1 #define MONITOR_ALL_NODES [0|1]

If enabled, activates all Manager functionality, including heartbeat and emergency monitoring of all nodes from node ID 1 to MAX_NR_OF_NODES.

8.8.2 #define USE_FULL_NODELIST [0|1]

If this is enabled, the Manager autonomously maintains a node list. It does so by using the highest SDO client channel to actively scan CANopen devices found in the network. The application can get access to the node list using the function MGR_GetNodeInfoPtr.

With the additional define of NODELIST_WITH_IDOBJECT the entire ID Object [1018h] with all 4 subentries is added to the node list.

8.8.3 #define NR_OF_HBCHECKS_PERCYCLE [num]

Defines the number of heartbeat lost checks that are made with each call to MGR_ProcessMgr(). A higher number ensures a faster response to nodes lost, however, requires verifying up to 127 timeouts with each call. Default is four.

8.8.4 #define NR_OF_SDO_CLIENTS [num]

Number of SDO or USDO client channels that the node can use at the same time. If auto-scan is used (USE_FULL_NODELIST==1), the highest channel number is reserved for the scanning and must not be used by the application. Default is 8 channels.

8.9 SDO settings of nodecfg.h

8.9.1 #define SDO_REQUEST_TIMEOUT [num]

Defines the default timeout used for SDO requests in milliseconds. After a request is sent, the timeout starts, if a node does not respond within the timeout, the manager assumes the node does not exist. Default is 50 milliseconds.

8.9.2 #define SDO_BACK2BACK_TIMEOUT [num]

Defines the timeout used between the transmissions of SDO requests in milliseconds. A Manager should itself not produce 100% bus load which could easily happen if many SDO configurations need to be done and when using multiple SDO channels in parallel. This timeout delays back to back transmits by the manager. Default is three milliseconds.

8.9.3 #define USE_BLOCKED_SDO_CLIENT [0|1]

If enabled, activates the SDO block-transfer support for the SDO client.

8.9.4 #define SDO_BLK_MAX_SIZE [4-127]

If block-transfer for the SDO client is enabled, this sets the maximum number of blocks allowed.

8.9.5 #define SDOCLNTCB_APPSDO_WRITE [0|1]

If enabled, allow SDO client writes/downloads of longer data than local buffer. Enables SDOCLNTCB_SDOWriteInit and SDOCLNTCB_SDOWriteComplete call-back functions.

8.10 USDO settings of nodecfg.h

8.10.1 #define USDO_REQUEST_TIMEOUT [num]

Defines the default timeout used for USDO requests in milliseconds. After a request is sent, the timeout starts, if a node does not respond within the timeout, the SDO client assumes the node does not exist. Default is 250 milliseconds.

8.10.2 #define USDO_B2BDELAY_MAX [num]

During download, the USDO client receives, from each server, a desired segment processing time. It honors these as long as they don't delay back-to-back messages more than the time given here (in milliseconds).

8.10.3 #define USDO_FLAGTYPE_64 [0|1]

The USDO client supports managing up to 64 server nodes during broadcast communication when the UNSIGNED64 (unsigned long long) data type is available, or 32 otherwise. Enable if the data type is available.

8.10.4 #define USDOCLNTCB_APPSDO_WRITE [0|1]

If enabled, allow USDO client writes/downloads of longer data than local buffer. Enables USDOCLNTCB_USDOWriteInit and USDOCLNTCB_USDOWriteComplete call-back functions.