



# **MICROCANOPEN PLUS USER MANUAL**

Revision 348 for Version 4.00 of MicroCANopen Plus

MICROCANOPEN PLUS COMMERCIAL LICENSE  
EMBEDDED SYSTEMS ACADEMY, INC.  
For MicroCANopen Plus V4.00

You should carefully read the following terms and conditions before using this software. Unless you have a different license agreement signed by Embedded Systems Academy, Inc. ("ESA") your use of this copy of MicroCANopen (the "SOFTWARE") indicates your acceptance of this license.

If you do not agree to any of the terms of this License, then do not use this copy of the SOFTWARE.

If the SOFTWARE is used for a project that is rented, leased, sold or otherwise traded (a "COMMERCIAL PROJECT") then this commercial license is required to use the SOFTWARE. If the SOFTWARE is used to develop knowledge of CANopen for a COMMERCIAL PROJECT then this commercial license is required.

This license is not free! TO USE THIS LICENSE YOU MUST PURCHASE A LICENSE FOR MICROCANOPEN PLUS OR A COPY OF "PCANopen Magic ProDS" FROM WWW.CANOPENSTORE.COM, PEAK-SYSTEM TECHNIK, GMBH, OR ONE OF IT'S DISTRIBUTORS.

Installation and Use. You may install and use an unlimited number of copies of the SOFTWARE.

Reproduction and Distribution. You may not reproduce and distribute copies of the SOFTWARE without written permission of ESA.

All title and copyrights in and to the SOFTWARE (including but not limited to any images, photographs, animations, video, audio, music, text, and "applets" incorporated into the SOFTWARE), any accompanying printed materials, and any copies of the SOFTWARE are owned by ESA. The SOFTWARE is protected by copyright laws and international treaty provisions. Therefore, you must treat the SOFTWARE like any other copyrighted material. All copyright notices, this license, header comments and similar statements include with this distribution of the SOFTWARE must remain in the source code at all times. No claim must be made as to the ownership of the SOFTWARE.

THIS SOFTWARE, AND ALL ACCOMPANYING FILES, DATA AND MATERIALS, ARE DISTRIBUTED "AS IS" AND WITH NO WARRANTIES OF ANY KIND, WHETHER EXPRESS OR IMPLIED. Good data processing procedure dictates that any program be thoroughly tested with non-critical data before relying on it. The user must assume the entire risk of using the program. THIS DISCLAIMER OF WARRANTY CONSTITUTES A ESSENTIAL PART OF THE AGREEMENT.

IN NO EVENT SHALL ESA, OR ITS PRINCIPALS, SHAREHOLDERS, OFFICERS, EMPLOYEES, AFFILIATES, CONTRACTORS, SUBSIDIARIES, OR PARENT ORGANIZATIONS, BE LIABLE FOR ANY INCIDENTAL, CONSEQUENTIAL, OR PUNITIVE DAMAGES WHATSOEVER RELATING TO THE USE OF THE SOFTWARE, OR YOUR RELATIONSHIP WITH ESA.

IN ADDITION, IN NO EVENT DOES ESA AUTHORIZE YOU TO USE THE SOFTWARE IN APPLICATIONS OR SYSTEMS WHERE THE SOFTWARE'S FAILURE TO PERFORM CAN REASONABLY BE EXPECTED TO RESULT IN A SIGNIFICANT PHYSICAL INJURY, OR IN LOSS OF LIFE. ANY SUCH USE BY YOU IS ENTIRELY AT YOUR OWN RISK, AND YOU AGREE TO HOLD ESA HARMLESS FROM ANY CLAIMS OR LOSSES RELATING TO SUCH UNAUTHORIZED USE.

This Agreement is the complete statement of the Agreement between the parties on the subject matter, and merges and supersedes all other or prior understandings, purchase orders, agreements and arrangements. This Agreement shall be governed by the laws of the State of California. Exclusive jurisdiction and venue for all matters relating to this Agreement shall be in courts and for a located in the State of California, and you consent to such jurisdiction and venue.

All rights of any kind in the SOFTWARE which are not expressly granted in this License are entirely and exclusively reserved to and by ESA.

## TABLE OF CONTENTS

TABLE OF CONTENTS .....	4
1 THE MICROCANOPEN PROTOCOL STACK.....	7
1.1 MICROCANOPEN AND MICROCANOPEN PLUS .....	7
1.2 MICROCANOPEN MANAGER ADD-ON.....	7
1.3 CANOPEN DOCUMENTATION.....	7
1.4 FILE AND DIRECTORY STRUCTURE .....	7
1.5 FUNCTION SUMMARY.....	10
1.6 DS401 GENERIC I/O EXAMPLE APPLICATION.....	11
1.7 CiA447 CAR ADD-ON DEVICES EXAMPLE APPLICATION.....	11
2 APPLICATION INTERFACE .....	13
2.1 THE PROCESS IMAGE .....	13
2.1.1 CONFIGURATION OF THE PROCESS IMAGE .....	13
2.2 OBJECT DICTIONARY CONFIGURATION .....	14
2.2.1 CONSTANT EXPEDITED OBJECT DICTIONARY ENTRIES .....	14
2.2.2 VARIABLE EXPEDITED AND MAPPED OBJECT DICTIONARY ENTRIES .....	15
2.2.3 GENERIC OBJECT DICTIONARY ENTRIES (PLUS).....	16
2.3 CANOPEN API FUNCTIONS.....	18
2.3.1 The MCO_Init function .....	18
2.3.2 The MCO_ReadProcessData function (PLUS) .....	19
2.3.3 The MCO_WriteProcessData function (PLUS).....	19
2.3.4 The MCO_InitRPDO function.....	20
2.3.5 The MCO_InitTPDO function.....	20
2.4 CANOPEN API CALL-BACK FUNCTIONS.....	21
2.4.1 The MCOUSER_NMTChange function (PLUS) .....	21
2.4.2 The MCOUSER_SYNCReceived function (PLUS).....	22
2.4.3 The MCOUSER_RPDOReceived function (PLUS).....	22
2.4.4 The MCOUSER_FatalError function .....	23
2.4.5 The MCOUSER_SDOResponse function (PLUS) .....	23

2.4.6	The MCOUSER_GetSerial function.....	24
2.5	CANOPEN API EXTENDED FUNCTIONS ( <i>PLUS</i> ).....	25
2.5.1	The MCOP_ProcessHBCheck function.....	25
2.5.2	The MCOSP_GetStoredParameters function .....	25
2.6	DRIVER FUNCTIONS .....	26
2.6.1	The MCOHW_Init function.....	26
2.6.2	The MCOHW_SetCANFilter function.....	26
2.6.3	The MCOHW_PushMessage function .....	26
2.6.4	The MCOHW_PullMessage function.....	27
2.6.5	The MCOHW_GetTime function.....	27
2.6.6	The NVOL_ReadByte function (Plus).....	27
2.6.7	The NVOL_WriteByte function (Plus).....	28
2.7	USING SOFTWARE CAN FILTERS AND FIFOS .....	28
2.7.1	USING SOFTWARE CAN RECEIVE FILTERS .....	28
2.7.2	USING THE FIFOS.....	29
2.7.3	SAMPLE CAN RECEIVE INTERRUPT IMPLEMENTATION.....	29
3	CANOPEN CODE CONFIGURATION.....	30
3.1	TABLE SIZE SETTINGS OF PROCIMG.H.....	30
3.1.1	#define PROC_IMGSIZE 96 .....	30
3.2	GENERAL SETTINGS OF NODECFG.H.....	30
3.2.1	(PLUS) #define USE_MCOP .....	30
3.2.2	#define CHECK_PARAMETERS .....	30
3.2.3	#define USE_LEDS .....	30
3.3	PDO SETTINGS OF NODECFG.H.....	30
3.3.1	#define NR_OF_RPDOS 2 .....	30
3.3.2	#define NR_OF_TPDOS 2 .....	31
3.3.3	#define USE_EVENT_TIME.....	31
3.3.4	#define USE_INHIBIT_TIME.....	31
3.3.5	(PLUS) #define USE_SYNC.....	31
3.4	NMT SERVICE SETTINGS OF NODECFG.H ( <i>PLUS</i> ).....	31
3.4.1	#define AUTOSTART .....	31
3.4.2	#define DEFAULT_HEARTBEAT.....	31

3.4.3	(PLUS) #define NR_HB_CONSUMER 0, (PLUS) #define DYNAMIC_HEARTBEAT_CONSUMER .....	32
3.4.4	(PLUS) #define USE_EMCY .....	32
3.4.5	(PLUS) #define USE_NODE_GUARDING .....	32
3.4.6	(PLUS) #define USE_STORE_PARAMETERS .....	32
3.4.7	(PLUS) #define NVOL_STORE_START 0.....	32
3.4.8	(PLUS) #define NVOL_STORE_SIZE 200 .....	32
3.4.9	(PLUS) #define USECB_NMTCHANGE.....	33
3.4.10	(PLUS) #define USECB_SYNCRECEIVE.....	33
3.4.11	(PLUS) #define USECB_RPDORECEIVE.....	33
3.4.12	(PLUS) #define USECB_SDOREQ .....	33
4	USING AUTO-GENERATED SOURCES .....	34
4.1	FILE GENERATION .....	34
4.2	FILE INTEGRATION .....	34
4.2.1	PIMG.H.....	34
4.2.2	INITPDOS.H.....	35
4.2.3	ENTRIESANDREPLIES.H.....	35

## **1 THE MICROCANOPEN PROTOCOL STACK**

The MicroCANopen protocol stack implements all mandatory functionality of the CiA (CAN in Automation user's and manufacturer's group) standard DS301 "CANopen Application Layer and Communication Profile" version 4.02 and selected functionality of the standard DSP302 "CANopen Framework for CANopen Managers and Programmable CANopen Devices" version 3.21. The examples included are in accordance to the standard DS401 "CANopen Device Profile for Generic I/O Modules" version 2.1.

The CiA447 version provides examples for the implementation of car add-on devices and includes the Manager functionality.

Examples implementations for other Device or Application Profiles are available upon request.

### **1.1 MICROCANOPEN AND MICROCANOPEN PLUS**

This manual covers MicroCANopen Plus, see [www.microcanopen.com](http://www.microcanopen.com) for details on the basic version. This "Plus" version is included with the Development System version of the CANopen configuration and test utility "CANopen Magic ProDS".

### **1.2 MICROCANOPEN MANAGER ADD-ON**

Advanced CANopen Manager functionality as defined in "CANopen Framework for CANopen Managers and Programmable CANopen Devices" is available as an add-on package to MicroCANopen Plus. This add-on is included in the CiA447 version of MicroCANopen. For details see the MicroCANopen Manager User Manual or [www.CANopenStore.com](http://www.CANopenStore.com).

### **1.3 CANOPEN DOCUMENTATION**

It is assumed that programmers using MicroCANopen have a general understanding about how CANopen works. In addition they should either have access to the CANopen specification or a CANopen book such as "Embedded Networking with CAN and CANopen" ([www.CANopenBook.com](http://www.CANopenBook.com)). The MicroCANopen manual does not explain regular CANopen features, functions and terms.

### **1.4 FILE AND DIRECTORY STRUCTURE**

The directory structure used by MicroCANopen separates the files used into four 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.

1.) Common Shared Directory:

Name: ../MCO

This directory contains all files implementing the core features of the CANopen protocol. 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.

File / Module	Content
mcohw.h	CAN Driver interface definition
mco.h mco.c	MicroCANopen core module
mcop.h mcop.c	Generic MicroCANopen Plus extensions of MicroCANopen
storpara.c	MicroCANopen Plus extension: Store Parameters
xsdo.h xsdo.c	MicroCANopen Plus extension: Segmented SDO transfers (Supporting Object Dictionary entries that are > 4 bytes)
lss.h lssslv.c	MicroCANopen Plus extension: Layer Setting Services, slave implementation
canfifo.h canfifo.c	Implementation of CAN software filtering and transmit and receive FIFOs
mlss.h mlssslv.h	MicroLSS slave implementation implementing the LSS FastScan

2.) Application Configuration Directories

Name: ../MCO\_ *APPLICATIONNAME*

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

File / Module	Content
mcohw XXX.h mcohwXXX.c	CANopen driver implementation. Provides CAN communication routines and timer handling.
mcohwLEDs.h	Defining access to optional RUN and ERR LEDs
procimg.h	Definition of symbolic offsets for locations in the process image
nodecfg.h	MicroCANopen functionality configuration

userXXX.c mainXXX.c	User specific CANopen code including Object Dictionary contents, PDO settings and call-back functions
------------------------	---

Name: ../MCO\_APPLICATIONNAME/EDS

This directory contains the application's EDS and DCF files (Electronic Data Sheet and Device Configuration File) as well as auto-generated source code files generated by the CANopen EDS Editor "CANopen Architect EDS".

File / Module	Content
Application.eds	Application's Electronic Data Sheet
Application.dcf	Application's Device Configuration File. This is for a specific node ID and is the file used as a basis to the auto-generated files below.
entriesandreplies.h initpdos.h pimg.h	Auto-generated configuration files generated by "CANopen Architect EDS" using the Device Configuration File above.

### 3.) Simulation Specific Directory

Name: ../MCO\_simulator

This directory contains the source files that are required when compiling MicroCANopen for the PCANopen Magic ProDS simulation environment.

### 4.) Optional Common Shared Directory:

Name: ../MGR

This directory contains all files implementing CANopen Manager functionality. This is only included in the delivery if the manager add-on option is ordered or the CiA447 car add-on devices version of the code.

File / Module	Content
comgr.h comgr.c	Implements the basic functionality of a Manager
lssmgr.h lssmgr.c	Optional LSS Master implementation
mlssmgr.h mlssmgr.c	Optional Micro LSS Master implementation with LSS Fast Scan
concisedcf.h concisedcf.c	Optional support of concise DCF

## 1.5 FUNCTION SUMMARY

MicroCANopen can be used to implement CANopen Slave nodes in accordance with almost any Device profile or Application profile available today. However, not all advanced CANopen functions defined by the standards are implemented. MicroCANopen Plus covers the advanced functionality most often used in CANopen slave nodes.

### OBJECT DICTIONARY

MicroCANopen and MicroCANopen Plus implement an object dictionary with one SDO server. The basic version is limited to expedited SDO transfers. This means that no single variable stored in the Object Dictionary can exceed 4 bytes. Longer variables must be divided into multiple 4-byte values. In addition, MicroCANopen Plus supports segmented SDO transfers allowing access to Object Dictionary entries of more than 4 bytes.

Using the SDO server, one Manager or configuration tool can send read/write requests to the Object Dictionary.

### HEARTBEAT VS. NODE GUARDING

As recommended by the CiA and other CANopen experts, MicroCANopen implements the newer heartbeat method instead of the older node guarding method. However, in order to better work with legacy devices (including the CANopen conformance test) MicroCANopen Plus also has a very basic version of node guarding implemented.

### MICROCANOPEN PDO PARAMETERS

In MicroCANopen all PDO parameters (communication and mapping) are hard-coded and cannot change during operation. PDO trigger options include time driven as well as event driven with an inhibit time.

### MICROCANOPEN PLUS PDO PARAMETERS

Using MicroCANopen Plus, the PDO communication parameters are dynamic and can change during operation – for example by overwriting the defaults with a configuration tool. Due to the usage of a very simple driver the PDO linking (CAN message ID used for each PDO) can only be changed until the device goes operational for the first time. As an additional trigger function, MicroCANopen Plus supports synchronized transfer of TPDOS (in reply to the SYNC message).

### NUMBER OF PDOS

The maximum number of PDOS supported are 1024 TPDOS and 1024 RPDOS for MicroCANopen Plus.

### EMERGENCY PRODUCER

MicroCANopen Plus supports 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.

#### HEARTBEAT CONSUMER

MicroCANopen Plus provides multiple heartbeat consumer channels. 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. So if the application knows which heartbeats it should listen to, then that information can be directly used without waiting for a configuration through the network.

#### STORE PARAMETERS

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

#### LAYER SETTING SERVICES

Using the layer setting services, MicroCANopen Plus based nodes can change their node ID and or the bit rate settings during operation. An LSS Master is required in the CANopen network to use this functionality.

Since V4.00 MicroCANopen Plus supports MicroLSS, the LSS Fast Scan service that auto-identifys non-configured nodes on the network.

#### USER CALL-BACK FUNCTIONS

MicroCANopen provides call-back functions for the resets and for fatal errors. The reset function is typically used to initialize the entire CANopen stack.

MicroCANopen Plus provides optional call-back functions for changes in the NMT Slave state machine and for handling unknown SDO requests to the Object Dictionary. The later can be used to implement very application specific Object Dictionary entries.

## **1.6 DS401 GENERIC I/O EXAMPLE APPLICATION**

The example code supplied with MicroCANopen implements a minimal DS401 compliant device with 4 digital input bytes, 4 digital output bytes, 2 analog input words and 2 analog output words. The process data is transmitted using 2 Transmit PDOs and 2 Receive PDOs.

The output data send to the application is directly echoed back as input data. Values send to RPDO1 are echoed back on TPDO1, values send to RPDO2 are echoed back on TPDO2.

## **1.7 CiA447 CAR ADD-ON DEVICES EXAMPLE APPLICATION**

The example code supplied with MicroCANopen Plus for CiA 447 implements several examples for car add-on devices. The devices come up non-configured

and wait for the gateway to detect and configure them using the MicroLSS Fast Scan detection cycle.

To manually configure the nodes and have them autostart with a fixed node ID one needs to disable the MicroLSS service in `nodecfg.h`. In that file simply comment out the defines for `USE_LSS_SLAVE` and `USE_MICROLSS`.

The node ID is no set by the DCF configuration. To change, modify the DCF belonging to the application (stored in `EDS_XXX` directory) using Code Architect EDS and re-generate the source files from there.

## 2 APPLICATION INTERFACE

Both shared data memory and function calls are used to implement an interface between MicroCANopen the application program. A process image (array of bytes) is used as shared memory that can be accessed from both MicroCANopen 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.

### 2.1 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.

#### 2.1.1 CONFIGURATION OF THE PROCESS IMAGE

Since version 2.6 the process image configuration can be automatically generated by CANopen Architect EDS. Example directories ending in “\_CA” contain examples that use such automatically generated configurations. The default file name for the file containing the process image variable definitions is *pimg.h*.

Where exactly each variable is located in the process image is part of the CANopen node configuration process that needs to be done by the designer/programmer of the CANopen node. The CANopen configuration process also includes assigning an Object Dictionary Index and Subindex to each variable and to configure the PDOs (Process Data Objects) containing one or multiple process data variables.

To simplify accessing the process image and to allow for easy re-configuration of process images, it is recommended to use *#define* statements to define the offsets to the individual variables in the process image. These should be defined in the

file *procing.h* that can be included to all code modules requiring access to the process image.

In MicroCANopen it MUST be ensured that all variables mapped into one PDO (one CAN message) are located consecutively in the process image. The entire contents of PDOs is copied byte-by-byte from/to the process image.

## ACCESSING THE PROCESS IMAGE

Any application program may directly access the data in the process image (for example:  $gProcImg[offset] = x$ ). The Macros  $PIxxACC()$  are defined to simplify access to 16 and 32 bit values:  $PI16ACC[offse\_to\_16bit\_valuet] = 0x1234$

For compatibility with other implementations and in order to potentially protect the data consistency and integrity, it is recommended to use the access functions provided by MicroCANopen. These functions are  $MCO\_ReadProcessData()$  and  $MCO\_WriteProcessData()$ .

## 2.2 OBJECT DICTIONARY CONFIGURATION

Since version 2.6 the Object Dictionary configuration can be automatically generated by CANopen Architect EDS. Example directories ending in “\_CA” contain examples that use such automatically generated configurations. The default file name for the file containing the process image variable definitions is *entriesandrepplies.h*.

Although working with CANopen EDS and DCF files is the standard procedure for many CANopen configuration tools, many embedded CANopen nodes require a specific default configuration that a node should use if not configured through a CANopen configuration tool or by a CANopen Configuration Manager.

In MicroCANopen the default configuration is setup via tables typically implemented in a file called *user\_XXX.c* (User Object Dictionary file).

The tables *gSDOResponseTable* and *gODProcTable* define the contents of the Object Dictionary. When using auto-generated files the auto-generated data can be included into these tables using the Macros  $SDOREPLY\_ENTRIES$ ,  $ODENTRY\_ENTRIES$  and  $ODGENTRY\_ENTRIES$ .

### 2.2.1 CONSTANT EXPEDITED OBJECT DICTIONARY ENTRIES

#### The *gSDOresponseTable* table

The table *gSDOresponseTable* is an array of bytes that contains a list of SDO responses for SDO requests to constant, read-only entries in the object dictionary limited to 4 bytes or less. Typically these contain the [1000,00] Device Type entry, the [1018,xx] Identity Objects and some “Number of Entries” type entries with a Subindex of zero.

Each entry in this list has 8 bytes that directly contain the 8 bytes used in a CAN message with an expedited SDO response to a read (upload) request.

The macros *SDOREPLY* and *SDOREPLY4* are provided to ease the generation of the 8-byte entries.

The last entry must be 8 times *0xFF* to indicate the end of the table.

The current implementation does not require that the entries are sorted in any way.

#### **The *SDOREPLY* macro**

This macro generates the 8-byte SDO response required for a read (upload) request from an Object Dictionary entry with a constant entry.

*SDOREPLY*(INDEX,SUBINDEX,LENGTH,VALUE)

INDEX is the 16-bit Index of the Object Dictionary entry.

SUBINDEX is the 8-bit Subindex of the Object Dictionary entry.

LENGTH is the length of the Object Dictionary entry in bytes and must be in the range of 1 to 4.

VALUE is the value of the Object Dictionary entry. It must be defined as a 32-bit value even if LENGTH is less than 4-bytes. In that case the unused bytes must be set to zero.

The Object Dictionary entry [1000h,00h] with a value of 00030191h can be generated by:

*SDOREPLY*(0x1000,0x00,4,0x00030191L),

#### **The *SDOREPLY4* macro**

This macro generates the 8-byte SDO response required for a read (upload) request from an Object Dictionary entry with a constant entry of 4 bytes with an ASCII interpretation. This simplifies the generation of 32-bit Object Dictionary entries whose contents are not interpreted as a 32-bit value but as 4 characters.

*SDOREPLY4*(INDEX,SUBINDEX,CHAR1,CHAR2,CHAR3,CHAR4)

INDEX is the 16-bit Index of the Object Dictionary entry.

SUBINDEX is the 8-bit Subindex of the Object Dictionary entry.

CHAR1 through CHAR4 contain the 4 characters stored at this Object Dictionary entry.

### **2.2.2 VARIABLE EXPEDITED AND MAPPED OBJECT DICTIONARY ENTRIES**

#### **The *gODProcTable***

This table is an array of structures that defines Object Dictionary entries whose data is located in the process image and that can be mapped into PDOs (Process

Data Objects). All Object Dictionary entries that can be mapped to a PDO or need to be shared with the application via the process image must be defined in this table. The macro *ODENTRY* can be used to simplify entries into this table.

The last entry must use the index FFFFh to mark the end of the table.

The current implementation does not require that the entries are sorted in any way.

**The ODENTRY macro**

*ODENTRY*(INDEX,SUBINDEX,TLINFO,OFFSET)

INDEX is the 16-bit Index of the Object Dictionary entry.

SUBINDEX is the 8-bit Subindex of the Object Dictionary entry.

TLINFO is an 8-bit value that defines access type and length of the Object Dictionary entry. The TLINFO value can be generated by adding up the length of the Object Dictionary entry (must be in the range of 1 to 4) and the following status bits:

- if the entry is readable via SDO requests, add *ODRD*
- if the entry is writable via SDO requests, add *ODWR*

Note that an entry can be both readable and writable.

OFFSET defines the location of the data for this Object Dictionary entry in the process image. If set to 3, the data is located starting at the 4<sup>th</sup> byte in the process image.

An Object Dictionary entry [6200h,01h] containing a one byte value that supports both read and write accesses and whose data is located in the 8<sup>th</sup> byte of the process image is defined as follows:

*ODENTRY*(0x6200,0x01,1+*ODRD*+*ODWR*,7),

**2.2.3 GENERIC OBJECT DICTIONARY ENTRIES (PLUS)**

**The gODGenericTable**

This table contains the remaining, generic Object Dictionary entries, typically longer than 4 bytes. This data may also be located outside the process image as it works with pointers that can point to any memory location. There are two macros provided. *ODGENTRYYP* is for entries that are located in the process image and *ODGENTRYYC* is used for entries using a pointer to any memory location (intended usage is for constant values, hence ‘C’).

**The ODGENTRYYP macro**

*ODGENTRYYP*(INDEX,SUBINDEX,ACCESS,LENGTH,OFFSET)

INDEX is the 16-bit Index of the Object Dictionary entry.

SUBINDEX is the 8-bit Subindex of the Object Dictionary entry.

ACCESS is an 8-bit value that defines the access type of the Object Dictionary entry. The following status bits are used:

- if the entry is readable via SDO requests, add *ODRD*
- if the entry is writable via SDO requests, add *ODWR*

Note that an entry can be both readable and writable.

LENGTH is the length of the Object Dictionary entry in bytes.

OFFSET defines the location of the data for this Object Dictionary entry in the process image. If set to 3, the data is located starting at the 4<sup>th</sup> byte in the process image.

An Object Dictionary entry [2200h,00h] containing a 10 byte value that supports both read and write accesses and whose data is located in the 8<sup>th</sup> byte of the process image is defined as follows:

*ODGENTRYP(0x2200,0x00,ODRD+ODWR,10,7),*

#### **The ODGENTRYC macro**

*ODGENTRYC(INDEX,SUBINDEX,ACCESS,LENGTH,POINTER)*

INDEX is the 16-bit Index of the Object Dictionary entry.

SUBINDEX is the 8-bit Subindex of the Object Dictionary entry.

ACCESS is an 8-bit value that defines the access type of the Object Dictionary entry. The following status bits are used:

- if the entry is readable via SDO requests, add *ODRD*
- if the entry is writable via SDO requests, add *ODWR*

Note that an entry can be both readable and writable.

LENGTH is the length of the Object Dictionary entry in bytes.

POINTER defines the location of the data for this Object Dictionary entry in the memory of the microprocessor.

An Object Dictionary entry [1008h,00h] containing a read-only string is defined as follows:

*ODGENTRYC(0x1008,0x00,ODRD,23,&(" MicroCANopen DS401 Demo")),*

## 2.3 CANOPEN API FUNCTIONS

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

### 2.3.1 *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 baudrate in kbit  
    UNSIGNED8 Node_ID, // CANopen node ID (1-126)  
    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:

0	use default or pre-defined 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, MicroCANopen will use the default or pre-configured node ID.

`Heartbeat` is the heartbeat producer time in milliseconds. If set to zero, MicroCANopen will try to use a default or pre-defined value.

#### **Returned**

Nothing.

### 2.3.2 *The MCO\_ReadProcessData function (PLUS)*

This function is used to read data from the process image.

#### **Declaration**

```
UNSIGNED8 MCO_ReadProcessData (  
    UNSIGNED8 *pDest, // Destination pointer  
    UNSIGNED8 length, // Number of bytes to copy  
    UNSIGNED8 offset // Offset of source data in process  
    image  
);
```

#### **Passed**

The pointer `pDest` is a destination pointer to the location to which the requested process data should be copied. The caller must ensure that the buffer at the destination locations is large enough to hold the number of data bytes requested.

The parameter `length` defines the number of data bytes requested.

The `offset` defines the location of the requested data within the process image. If set to zero, the data is located at the first byte of the process image.

#### **Returned**

The number of bytes actually copied to the destination buffer is returned. If zero, no data was copied because the requested offset was out of range.

### 2.3.3 *The MCO\_WriteProcessData function (PLUS)*

This function is used to write data to the process image.

#### **Declaration**

```
UNSIGNED8 MCO_WriteProcessData (  
    UNSIGNED8 offset, // Offset, destination in process image  
    UNSIGNED8 length, // Number of bytes to copy  
    UNSIGNED8 *pSrc // Source pointer  
);
```

#### **Passed**

The parameter `offset` defines the location of the target data within the process image. If set to zero, the data is located at the first byte of the process image.

The `length` defines the number of data bytes to be copied.

The pointer `pSrc` is a source pointer to the location from which the process data should be copied.

#### **Returned**

The number of bytes actually copied to the process image is returned. If zero, no data was copied because the requested offset was out of range.

### 2.3.4 *The MCO\_InitRPDO function*

This function initializes a Receive Process Data Object.

#### **Declaration**

```
void MCO_InitRPDO (  
    UNSIGNED8 PDO_NR,    // RPDO number (starting at 1)  
    UNSIGNED16 CAN_ID,   // CAN identifier (0 for default)  
    UNSIGNED8 len,       // Number of data bytes in RPDO  
    UNSIGNED8 offset     // Offset to data in process image  
)
```

#### **Passed**

The parameter `PDO_NR` defines the PDO number as used in CANopen. The default PDOs of a CANopen device are numbered 1 through 4.

The `CAN_ID` specifies the CAN message identifier used for this PDO. If left at zero the CANopen default is used.

The `len` parameter defines the number of data bytes in the PDO.

The parameter `offset` defines the location of the PDO data within the process image.

#### **Returned**

Nothing.

### 2.3.5 *The MCO\_InitTPDO function*

This function initializes a Transmit Process Data Object.

#### **Declaration**

```
void MCO_InitTPDO (  
    UNSIGNED8 PDO_NR,    // TPDO number (starting at 1)  
    UNSIGNED16 CAN_ID,   // CAN ID to use (0 for default)  
    UNSIGNED16 event_time, // Send every event_time ms  
    UNSIGNED16 inhibit_time, // Inhibit time in ms  
    // (set to 0 if ONLY event_time should be used)  
    UNSIGNED8 len,       // Number of data bytes in TPDO  
    UNSIGNED8 offset     // Offset to data in process image
```

**Passed**

The parameter `PDO_NR` defines the PDO number as used in CANopen. The default PDOs of a CANopen device are numbered 1 through 4.

The `CAN_ID` specifies the CAN message identifier used for this PDO. If left at zero the CANopen default is used.

The `event_time` defines how often this PDO is transmitted. This message is sent every `event_time` milliseconds.

The `inhibit_time` activates change-of-state transmission (transmission when data to be transmitted actually changed) and defines the minimum delay before a message can be transmitted again. Even if the state changes, the message is not re-transmit before `inhibit_time` expires.

The `len` parameter defines the number of data bytes in the PDO.

The parameter `offset` defines the location of the PDO data within the process image.

**Returned**

Nothing.

## 2.4 CANOPEN API CALL-BACK FUNCTIONS

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

### 2.4.1 *The MCOUSER\_NMTChange function (PLUS)*

This MicroCANopen 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.

**2.4.2 The MCOUSER\_SYNCReceived function (PLUS)**

This MicroCANopen Plus function is only available when the compiler directive USECB\_SYNCRECEIVE is defined. The function signals the receipt of the CANopen SYNC message for this device. Synchronous RPDO data previously received and copied to the process image may now be applied to the application. Per default configuration synchronous TPDO data transmission will be triggered after execution of this call-back function.

**Declaration**

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

**Passed**

Nothing.

**Returned**

Nothing.

**2.4.3 The MCOUSER\_RPDOReceived function (PLUS)**

This MicroCANopen Plus function is only available when the compiler directive USECB\_RPDORECEIVE is defined. The function signals the receipt of a Receive Process Data Object.

**Declaration**

```
void MCOUSER_RPDOReceived (  
    UNSIGNED8 RPDONr, // RPDO Number  
    UNSIGNED8 *pRPDO, // Pointer to RPDO data  
    UNSIGNED8 len     // Length of RPDO  
);
```

**Passed**

The parameters passed include the number of the RPDO (starting at 1), a pointer to the RPDO data (location in process image) and the number of bytes that were received with the RPDO.

**Returned**

Nothing.

**2.4.4 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.

**2.4.5 The *MCOUSER\_SDORequest* function (PLUS)**

This MicroCANopen Plus function is only available when the compiler directive `USECB_SDOREQ` is defined. The function signals the receipt of an SDO request for which MicroCANopen Plus has no answer. The Object Dictionary entry requested is NOT available.

This can be used to implement custom, manufacturer specific Object Dictionary entries. If this function works on the data received, it must generate the appropriate SDO response message or an SDO Abort. To do so, it may use functions provided by MicroCANopen: the function `MCO_ReplyWith` can be used to generate a response, the function `MCO_SendSDOAbort` to generate an abort.

**Declaration**

```
UNSIGNED8 MCOUSER_SDORequest (  
    UNSIGNED8 SDO_Data[8]  
);
```

**Passed**

SDO\_Data contains the 8 data bytes with the SDO request.

**Returned**

A return value of 0 indicates that MCOUSER\_SDORequest generated an SDO Abort message.

A return value of 1 indicates that MCOUSER\_SDORequest generated an SDO response.

A return value of 2 indicates that MCOUSER\_SDORequest does not implement any functionality for the Object Dictionary entry specified.

**2.4.6 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 32bit serial number of the device.

## 2.5 CANOPEN API EXTENDED FUNCTIONS (*PLUS*)

This section lists all functions considered extended functionality, only available in the Plus version of MicroCANopen. They typically require that certain define values are set to enable the functionality requested.

### 2.5.1 *The MCOP\_ProcessHBCheck function*

When heartbeat consumer functionality is enabled, this function verifies if a timeout occurred with any of the heartbeats consumed.

#### **Declaration**

```
UNSIGNED8 MCOP_ProcessHBCheck (  
    UNSIGNED8 consumer_channel  
)
```

#### **Passed**

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

#### **Returned**

Zero, if channel is not in use.

One, if channel is in use but monitoring did not yet start.

Two, if channel is in use and monitoring is active.

127, if a heartbeat timeout occurred, the node monitored by this channel did not transmit its heartbeat in time.

### 2.5.2 *The MCOSP\_GetStoredParameters function*

When the Store Parameters functionality is enabled, this function checks if the non-volatile memory contains any stored data. If it does, it retrieves the data and applies it. This function may only be called after MicroCANopen and all PDOs have been initialized.

#### **Declaration**

```
void MCOSP_GetStoredParameters (  
    void  
) ;
```

#### **Passed**

Nothing.

#### **Returned**

Nothing.

## 2.6 DRIVER FUNCTIONS

This section lists all functions that need to be provided by the driver level. If MicroCANopen is used on a microcontroller architecture for which there is no example included, then these functions must be implemented on the driver level and provided to for MicroCANopen.

### 2.6.1 *The MCOHW\_Init function*

```

/*****
DOES:   This function implements the initialization of the CAN
        interface.
RETURNS: 1 if init is completed
        0 if init failed
*****/
UNSIGNED8 MCOHW_Init (
    UNSIGNED16 BaudRate
    // Allowed values: 1000, 800, 500, 250, 125, 50, 25, 10
);

```

### 2.6.2 *The MCOHW\_SetCANFilter function*

```

/*****
DOES:   This function implements the initialization of a CAN ID hardware
        filter as supported by many CAN controllers.
RETURNS: 1 if filter was set
        2 if this HW does not support filters
        (in this case HW will receive EVERY CAN message)
        0 if no more filter is available
*****/
UNSIGNED8 MCOHW_SetCANFilter (
    UNSIGNED16 CANID // CAN-ID to be received by filter
);

```

### 2.6.3 *The MCOHW\_PushMessage function*

```

/*****
DOES:   This function implements a CAN transmit queue. With each
        function call a message is added to the queue.
RETURNS: 1 Message was added to the transmit queue
        0 If queue is full, message was not added,
NOTES:  The MicroCANopen stack will not try to add messages to the queue
        "back-to-back". With each call to MCO_ProcessStack, a maximum
        of one message is added to the queue. For many applications
        a queue with length "1" will be sufficient. Only applications
        with a high busload or very slow bus speed might need a queue
        of length "3" or more.
*****/
UNSIGNED8 MCOHW_PushMessage (
    CAN_MSG MEM_FAR *pTransmitBuf // Data structure with message to be send
);

```

### 2.6.4 *The MCOHW\_PullMessage function*

```

/*****
DOES:   This function implements a CAN receive queue. With each
        function call a message is pulled from the queue.
RETURNS: 1 Message was pulled from receive queue
         0 Queue empty, no message received
NOTES:   Implementation of this function greatly varies with CAN
        controller used. In an SJA1000 style controller, the hardware
        queue inside the controller can be used as the queue.
        Controllers with just one receive buffer need a bigger software
        queue. "Full CAN" style controllers might just implement
        multiple message objects, one each for each ID received (using
        function MCOHW_SetCANFilter).
*****/
UNSIGNED8 MCOHW_PullMessage (
    CAN_MSG MEM_FAR *pTransmitBuf // Data structure with message received
);

```

### 2.6.5 *The MCOHW\_GetTime function*

```

/*****
DOES:   This function reads a 1 millisecond timer tick. The timer tick
        must be a UNSIGNED16 and must be incremented once per
        millisecond.
RETURNS: 1 millisecond timer tick
NOTES:   Data consistency must be insured by this implementation.
        (On 8-bit systems, disable the timer interrupt incrementing
        the timer tick while executing this function)
        Systems that cannot provide a lms tick may consider incrementing
        the timer tick only once every "x" ms, if the increment is by
        "x".
*****/
UNSIGNED16 MCOHW_GetTime (
    void
);

```

### 2.6.6 *The NVOL\_ReadByte function (Plus)*

This function is only needed when the Store Parameter functionality is used.

```

/*****
DOES:   Reads a data byte from non-volatile memory.
RETURNS: The data read from memory
*****/
UNSIGNED8 NVOL_ReadByte (
    UNSIGNED16 address // location of byte in NVOL memory
);

```

### 2.6.7 *The NVOL\_WriteByte function (Plus)*

This function is only needed when the Store Parameter functionality is used.

```

/*****
DOES:    Writes a data byte to non-volatile memory
RETURNS: nothing
*****/
void NVOL_WriteByte (
    UNSIGNED16 address, // location of byte in NVOL memory
    UNSIGNED8 data
);

```

## 2.7 USING SOFTWARE CAN FILTERS AND FIFOS

For maximum portability to various CAN controllers the module *canfifo* implements CAN message receive filters in software including message FIFOs. Applications requiring Manager functionality like listening to ALL Heartbeats, Emergencies or SDO channels should use this module.

To activate the module, define USE\_CANFIFO and specify the following sizes:

```
#define TXFIFOSIZE X
```

The value X must be 0, 4, 8, 16 or 32. Setting this to one of the non-zero values implements a software transmit FIFO. Any CAN message transmitted is copied to this FIFO first. The default behavior is that the FIFO is checked for transmission by each 1ms timer interrupt.

```
#define RXFIFOSIZE Y
```

The value Y must be 0, 4, 8, 16 or 32. Setting this to one of the non-zero values implements a software receive FIFO. Any CAN message received is copied by the receive interrupt service routine to this FIFO first. Using the *MCOHW\_PullMessage()* function MicroCANopen periodically checks if a message arrived and requires processing.

```
#define MGRFIFOSIZE Z
```

The value Z must be 0, 4, 8, 16 or 32. Setting this to one of the non-zero values implements a software receive FIFO for CAN messages received by the Manager functionality. These are all Heartbeats, Emergencies and SDO Client responses. Any CAN message received for this is copied by the receive interrupt service routine to this FIFO first. Using the *MCOHWMGR\_PullMessage()* function MicroCANopen periodically checks if a message arrived and requires processing.

### 2.7.1 USING SOFTWARE CAN RECEIVE FILTERS

When this module is used an array of 2048 bits is used to set which CAN messages with which CAN message ID are received and processed. One bit represents one of the 2048 possible CAN IDs.

By a call to *CANSWFILTER\_Set(CAN\_ID)* the corresponding bit is set – the message with *CAN\_ID* is now received.

With a call to *CANSWFILTER\_Match(CAN\_ID)* if the corresponding bit is set and if the message needs to be received.

### 2.7.2 USING THE FIFOS

Each FIFO has its own access functions. To copy a CAN message into the FIFO, the function *CANxxxFIFO\_GetInPtr()* must be called. It returns a pointer to a structure *CAN\_MSG* to which the CAN message can now be copied. If a null pointer is returned the FIFO is full and a FIFO overrun needs to be signaled to the application that this message is now lost.

Once copying is completed, the function *CANxxxFIFO\_InDone()* must be called to update the internal FIFO in and out counters.

To read a message from the FIFO, the function *CANxxxFIFO\_GetOutPtr()* must be called. It returns a null pointer if the FIFO is empty. If at least one message is in the FIFO, then a pointer to the *CAN\_MSG* structure in the FIFO is returned. Data can now be retrieved using this pointer.

Once the message is fully retrieved, the function *CANTXFIFO\_OutDone()* must be called to update the internal FIFO in and out pointers.

### 2.7.3 SAMPLE CAN RECEIVE INTERRUPT IMPLEMENTATION

If a CAN controller is configured to receive ALL CAN messages on the bus, then the following steps need to be taken in the CAN receive interrupt:

- 1.) Check if CAN message ID is for Manager functionality  
Heartbeat, Emergency, SDO Response
- 2.) If yes, copy message to MGR FIFO and leave interrupt  
*CANMGRFIFO\_GetInPtr()*, copy data, *CANMGRFIFO\_InDone()*  
Note: On FIFO overrun report overrun to status variable
- 3.) Check if CAN message ID is for this CANopen node  
*CANSWFILTER\_Match(CAN\_ID)*
- 4.) If yes, copy message to RX FIFO and leave interrupt  
*CANRXFIFO\_GetInPtr()*, copy data, *CANRXFIFO\_InDone()*  
Note: On FIFO overrun report overrun to status variable

## 3 CANOPEN CODE CONFIGURATION

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

Since Version 2.6 MicroCANopen Plus source code files can automatically be generated by the CANopen Architect EDS.

### 3.1 TABLE SIZE SETTINGS OF PROCIMG.H

#### 3.1.1 *#define PROC\_IMAGESIZE 96*

This value specifies the total size of the process image in bytes. It must not exceed 0xFFEh. The value is

### 3.2 GENERAL SETTINGS OF NODECFG.H

#### 3.2.1 (PLUS) *#define USE\_MCOP*

This define enables the “Plus” functionality of MicroCANopen Plus. It must be defined when any of the functions in the module *mcop.c* are used.

#### 3.2.2 *#define CHECK\_PARAMETERS*

If *CHECK\_PARAMETERS* is defined, 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.

#### 3.2.3 *#define USE\_LEDS*

Defining *USE\_LEDS* enables two CANopen indicator lights as specified by the CiA document DR303. Both a RUN and ERR light are supported. When using this define, 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.

### 3.3 PDO SETTINGS OF NODECFG.H

#### 3.3.1 *#define NR\_OF\_RPDOS 2*

This value defines the number of RPDOS (Receive Process Data Objects) implemented. The value range may be from 0 to 16.

### 3.3.2 *#define NR\_OF\_TPDOS 2*

This value defines the number of TPDOs (Transmit Process Data Objects) implemented. The value range may be from 0 to 16.

### 3.3.3 *#define USE\_EVENT\_TIME*

If *USE\_EVENT\_TIME* is defined, TPDO trigger events may include using the event timer (periodic transmission every X milliseconds).

### 3.3.4 *#define USE\_INHIBIT\_TIME*

If *USE\_INHIBIT\_TIME* is defined, 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, MicroCANopen automatically does a divide or multiply by 10 when communicating the inhibit time via SDO requests/responses.

### 3.3.5 *(PLUS) #define USE\_SYNC*

If *USE\_SYNC* is defined, the TPDOs 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.

NOTE:

If synchronized RPDOs are desired, the call-back function *MCOUSER\_SYNCReceived* can be used as the indication that the output data stored in the process image can now be applied to the application.

## 3.4 NMT SERVICE SETTINGS OF NODECFG.H *(PLUS)*

### 3.4.1 *#define AUTOSTART*

When *AUTOSTART* is defined 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.

### 3.4.2 *#define DEFAULT\_HEARTBEAT*

The Object Dictionary entry [1017h,00h] Heartbeat Producer Time is implemented as read-write. The *DEFAULT\_HEARTBEAT* defines the default heartbeat time used by MicroCANopen and is specified in milliseconds.

**3.4.3 (PLUS) #define NR\_HB\_CONSUMER 0,  
(PLUS) #define DYNAMIC\_HEARTBEAT\_CONSUMER**

This value defines if the heartbeat consumer functionality is provided. 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.

When DYNAMIC\_HEARTBEAT\_CONSUMER is defined, 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.

**3.4.4 (PLUS) #define USE\_EMCY**

When USE\_EMCY is defined, MicroCANopen 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.

**3.4.5 (PLUS) #define USE\_NODE\_GUARDING**

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 MicroCANopen Plus supports minimal node guarding functionality that is enabled if this compiler directive is defined.

**3.4.6 (PLUS) #define USE\_STORE\_PARAMETERS**

When USE\_STORE\_PARAMETERS is defined, the Store Parameters functionality of MicroCANopen Plus is enabled. The module `storpara.c` must be integrated into the project.

**3.4.7 (PLUS) #define NVOL\_STORE\_START 0**

When USE\_STORE\_PARAMETERS is defined, 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.

**3.4.8 (PLUS) #define NVOL\_STORE\_SIZE 200**

When USE\_STORE\_PARAMETERS is defined, the define NVOL\_STORE\_SIZE must be set to the available size of the non-volatile memory. 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.

#### **3.4.9 (PLUS) #define USECB\_NMTCHANGE**

When USECB\_NMTCHANGE is defined, MicroCANopen Plus uses the call-back function MCOUSER\_NMTChange to signal a change in the NMT Slave State to the application.

#### **3.4.10 (PLUS) #define USECB\_SYNCRECEIVE**

When USECB\_SYNCRECEIVE is defined, MicroCANopen Plus uses the call-back function MCOUSER\_SYNCReceived to signal the reception of the SYNC signal to the application.

#### **3.4.11 (PLUS) #define USECB\_RPDORECEIVE**

When USECB\_RPDORECEIVE is defined, MicroCANopen Plus uses the call-back function MCOUSER\_RPDORReceived to signal the reception of an RPDO to the application.

#### **3.4.12 (PLUS) #define USECB\_SDOREQ**

When USECB\_SDOREQ is defined, MicroCANopen Plus uses the call-back function MCOUSER\_SDORequest to signal the reception of an unknown SDO request to the application.

## 4 USING AUTO-GENERATED SOURCES

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

The application examples provided with MicroCANopen Plus have their EDS, DCF and auto-generated files stored in the directory  
“`../MCO_APPLICATIONNAME/EDS`”

### 4.1 FILE GENERATION

When editing an EDS or DCF with CANopen Architect EDS 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 EDS 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 EDS then places these into the SDO Reply table and not into the process image.

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

### 4.2 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.

#### 4.2.1 *PIMG.H*

The file *pimg.h* contains the basic #define settings required by MicroCANopen 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.

### 4.2.2 *INITPDOS.H*

The file *initpdos.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`.

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 usage is:

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

### 4.2.3 *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\_xxx.c* file.

Usage 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[] =
{
    OENTRY_ENTRIES
    // End-of-table marker
    OENTRY(0xFFFF,0xFF,0xFF,0xFFFF)
};
```

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