MICROCANOPEN PLUS USER MANUAL

Revision 1700 for Version 5.50 of MicroCANopen Plus

MICROCANOPEN PLUS LICENSE AGREEMENT
EMBEDDED SYSTEMS ACADEMY, INC.
For MicroCANopen Plus V5.50

The enclosed software and documentation are the exclusive property of Embedded Systems Academy, Inc. (ESA), protected under the copyrights laws of the United States of America and under international treaty provisions. ESA agrees to grant LICENSEE a license to use this copy of MicroCANopen Plus (the "SOFTWARE") and LICENSEE agrees to pay for this license in accordance to the terms specified.

You should carefully read the following terms and conditions before using the SOFTWARE. Unless you have a different license agreement signed by ESA your use of 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.

Installation and Use. You may install and use an unlimited number of copies of the SOFTWARE for a SINGLE PROJECT at a SINGLE SITE within your organization.

Reproduction and Distribution. You may not reproduce and distribute copies of the SOFTWARE source code or parts thereof without written permission of ESA. You may freely reproduce and distribute binary firmware compiled using the SOFTWARE without limitations.

Third-party access. The SOFTWARE source code may be accessible only to employees within your organization at the SINGLE SITE that the SOFTWARE is licensed for. Any access from third-party personnel (consultants, temporary workers, clients, customers) or personnel not on site requires the written permission of ESA.

Limitations for Consultants, Software Houses, Product Development companies and other vendors of embedded development services: The SOFTWARE is licensed on a per-client basis. The restrictions to third-party access apply to clients or customers as well.

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.

Table of Contents

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 CiA 301 "CANopen Application Layer and Communication Profile" version 4.02 and selected functionality of the standard CiA 302 "CANopen Framework for CANopen Managers and Programmable CANopen Devices" version 3.21. The examples included are in accordance to the standard CiA 401 "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 for details on the basic version. This "Plus" version can be ordered separately but is also included with the bundle "CANopen Development Kit".

1.2 MICROCANOPEN MANAGER ADD-ON

Advanced CANopen Manager functionality as defined in CiA 302 "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.

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). 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 conform 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: support of the Store Parameters functions storing configuration data in non-volatile memory
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
dynpdo.c	Dynamic PDO Mapping functionality (add-on)

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 and process image access macros

nodecfg.h	MicroCANopen functionality configuration
user*XXX*.c mainXXX.c user_cbdata.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". He auto-generated files should not be modified as any re-generation of the files by CANopen Architect EDS would overwrite any modifications made manually.

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 stackinit.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 ProS 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.

PROCESS IMAGE USAGE
All data communicated via CANopen is organized in a process image, an array of bytes (type UNSIGNED8). Data is referred to by an offset into the process image. These offsets can be auto-generated by the CANopen configuration tool CANopen Architect EDS.

OBJECT DICTIONARY AND SDO SERVER
MicroCANopen Plus implements an object dictionary with one or multiple SDO servers. In basic configuration, the SDO server 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 can be configured to support segmented SDO transfers as well as SDO block transfer, allowing access to Object Dictionary entries longer 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. Note that this needs support of the underlaying CAN driver to receive RTR frames. This may not be readily available in every supported architecture or not even possible with every CAN controller.

MICROCANOPEN PDO PARAMETERS
In MicroCANopen Plus, PDO parameters may be static (hard-coded, not changeable during operation) or dynamic (changed during operation), depending on configuration. PDO trigger options include change-of-state with an inhibit time, event timer (periodical) and SYNC.
Due to the usage of a very simple driver API, the PDO linking (CAN message ID

used for each PDO) can only be changed until the device goes operational for the first time.

NUMBER OF PDOS

The maximum number of PDOs supported are 512 TPDOs and 512 RPDOs for MicroCANopen Plus. This limit is not a MicroCANopen limit, but the limit as specified by the CANopen standard.

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 then expected.

EMERGENCY CONSUMER

MicroCANopen Plus supports the consuming of emergency messages with the manager add-on package. When used, all 127 emergencies can be received and trigger a call-back function to the application.

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-identifies non-configured nodes on the network.

SDO FULLY-MESHED COMMUNICATION

For small networks of up to 16 nodes, MicroCANopen Plus features a built-in configuration that allows any node to access all of the object dictionary of another node at any time without restrictions. This is done by enabling 16 SDO server channels and 16 SDO clients in each node and setting the channels up using a custom COB-ID assignment scheme.

USER CALL-BACK FUNCTIONS

MicroCANopen provides call-back functions for the resets and for fatal errors. The communication 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.

All process image data accesses are made using macros. If such accesses need to be protected / locked from each other, then the macros can be used to include such locking calls.

1.6 DS401 GENERIC I/O EXAMPLE APPLICATION

The example code supplied with MicroCANopen implements a minimal CiA 401 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 now 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. The default file name for the file containing the process image variable definitions generated by CANopen Architect EDS 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 *procimg.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.

2.1.2 ACCESSING THE PROCESS IMAGE

Any application program may directly access the data in the process image (for example: *gProcImg[offset] = x*).

For a more generic access it is recommended to use the access functions and macros provided by MicroCANopen. See chapter 2.3.6 Process Image Access Macros: The PI_READ macro and below for details.

2.1.3 DATA INTEGRITY OF THE PROCESS IMAGE IN AN RTOS ENVIRONMENT

The process image is accessed by both the application and MicroCANopen (both with SDO and PDO accesses). If the MicroCANopen stack and the application cannot interrupt each other, then process image integrity is ensured and no further protection is required. This is true if both the application is running from a polling loop, such as in a main while(1) loop.

If MicroCANopen is used in a multi-task implementation, it needs to be ensured that accesses to the process image are not made “simultaneously” from multiple tasks. A Mutex or single token semaphore should be used that only one instance can access the process image at any given time.

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.

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.

2.2 OBJECT DICTIONARY CONFIGURATION

Since version 2.6, the Object Dictionary configuration can be automatically generated by CANopen Architect EDS. The default file name for the file containing the process image variable definitions is *entriesandreplies.h*.

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

For more details about the manual, advanced configuration of these tables see chapter 6.2 OBJECT DICTIONARY CONFIGURATION.

2.3 CANOPEN API FUNCTIONS AND MACROS

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 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, MicroCANopen will use the default or preconfigured node ID.

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

Returned

Nothing.

2.3.2 The MCO_InitRPDO function

This function initializes a Receive Process Data Object.
When using the code generation of CANopen Architect EDS these calls are automatically generated and provided in file *initpdos.h*.

Declaration

```
void MCO_InitRPDO (
  UNSIGNED16 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.

2.3.3 The MCO_InitTPDO function

This function initializes a Transmit Process Data Object.
When using the code generation of CANopen Architect EDS these calls are automatically generated and provided in file *initpdos.h*.

Declaration

```
void MCO_InitRPDO (
  UNSIGNED16 PDO_NR,     // TPDO number (starting at 1)
  UNSIGNED16 CAN_ID,     // CAN ID to use (0 for default)
  UNSIGNED16 event_time, // Send every event_tim ms
  UNSIGNED16 inhibit_time, // Inhibit time in ms
  // (set to 0 if ONLY event_tim 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.

2.3.4 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.

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

2.3.5 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.

2.3.6 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 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.

2.3.7 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 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.

2.3.8 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 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.

2.3.9 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

2.3.10 Macros for PDO process image access

For all PDO-related accesses, MicroCANopen 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.

2.3.11 The MCO_ReadProcessData function

This function can be 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.

Notes

The function makes use of the PI_READ macro (see chapter 2.3.6).

2.3.12 The MCO_WriteProcessData function

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.

Notes

The function makes use of the PI_WRITE macro (see chapter 2.3.7).

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_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.

2.4.2 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.

2.4.3 The MCOUSER_NMTChange function

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 "preoperational" entered

Returned

Nothing.

2.4.4 The MCOUSER_SYNCReceived function

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. 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.

2.4.5 The MCOUSER_RPDOReceived function

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.

Notes

For a synchronous (SYNC triggered) RPDO, this function is called after the SYNC has been received.

2.4.6 The MCOUSER_ODData function

This MicroCANopen 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 transfer..

Declaration

```
void MCOUSER_ODData (
  UNSIGNED16 idx,
  UNSIGNED8 subidx,
  UNSIGNED8 *pDat,
  UNSIGNED8 len
  );
```

Passed

The parameters passed include the 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.

2.4.7 The MCOUSER_TPDORdy function

This MicroCANopen Plus function is only available when the compiler directive USECB_TPDORDY is defined. The stack calls the function right before it sends a TPDO. In case the trigger was not under the application's control, such as in case of an expired event timer or a received SYNC, this function is called before the stack retrieves the TPDO-mapped data entries from the process image. This allows the application to update the TPDO data before it is sent, if necessary.

Declaration

```
void MCOUSER_TPDOReady (
  UNSIGNED16 TPDONr,       // TPDO Number
  UNSIGNED8  TPDOTrigger  // Trigger for this TPDO's
                          // transmission:
                          // 0: Event Timer
                          // 1: SYNC
                          // 2: SYNC+COS
                          // 3: COS or application trigger
  );
```

Passed

The parameters include the number of the TPDO (starting at 1) and the trigger that caused the TPDO to be sent.

Returned

Nothing.

Notes

Updating the TPDO-mapped data in the process image won't have any effect for TPDOs that were triggered because of change-of-state detection or manually from the application. In these cases the TPDO will be sent with the data that was in the process image before the call-back function. Further processing will use the updated data, however.

2.4.8 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.

2.4.9 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

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_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.

2.5.2 *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.

2.5.3 *The MCOP_GetStoredParameters function*

When the Store Parameters functionality is enabled, this function checks if the non-volatile memory 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.

2.5.4 The MCOP_PushEMCY function

When Emergency usage is enabled (*#define USE_EMCY*) functionality is enabled, a CANopen Emergency message can be transmitted with this function call.

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

Up to 5 bytes of manufacturer specific emergency / error information.

Returned

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

2.5.5 The MCOP_TransmitSleepObjection() function

When sleep mode usage is enabled (*#define USE_SLEEP*), this function can be called to transmit the sleep objection message. It should only be called when a sleep request was received by `MCOUSER_Sleep()`.

Declaration

```
void MCOP_TransmitSleepObjection
  ( void
  );
```

Passed

Nothing.

Returned

Nothing.

2.6 CANOPEN API EXTENDED CALLBACKS

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

2.6.1 The MCOUSER_AppSDOReadInit function

This MicroCANopen Plus function is only available when the compiler directive USECB_APPSDO_READ is defined. The function can be used to implement custom Object Dictionary read entries of any length. Data is transferred in segmented mode or block mode if activated.

Declaration

```
UNSIGNED8 MCOUSER_AppSDOReadInit (
  UNSIGNED8 sdoserver, // SDO server number
  UNSIGNED16 idx, // Index of OD entry
  UNSIGNED8 subidx, // Subindex of OD entry
  UNSIGNED32 *size, // RETURN: size of data
  UNSIGNED8 **pDat // RETURN: pointer to data buffer
  );
```

Passed

The parameters passed include the SDO server number (in range from zero to NR_OF_SDOSERVER-1), the idx (index) and subidx (subindex) of the Object Dictionary and the return values being the data size and a pointer pDat to the data.

Returned

0: The specified OD entry is not handled by this function.

1: The specified OD entry is handled by this function. A valid pointer and data size are returned.

5: An SDO abort SDO_ABORT_WRITEONLY is generated.

2.6.2 The MCOUSER_AppSDOWriteInit function

This MicroCANopen Plus function is only available when the compiler directive USECB_APPSDO_WRITE is defined. The function can be used to implement custom Object Dictionary write entries of any length. Data is transferred in segmented mode or block mode if activated. Calls to MCOUSER_AppSDOWriteComplete() are made if the transfer is completed or the destination buffer is full and more data is about to be received.

Declaration

```
UNSIGNED8 MCOUSER_AppSDOWriteInit (
  UNSIGNED8 sdoserver, // SDO server number
  UNSIGNED16 idx, // Index of OD entry
  UNSIGNED8 subidx, // Subindex of OD entry
  UNSIGNED32 *size, // RETURN: size of data
  UNSIGNED8 **pDat // RETURN: pointer to data buffer
  );
```

Passed

The parameters passed include the SDO server number (in range from zero to NR_OF_SDOSERVER-1), the idx (index) and subidx (subindex) of the Object Dictionary and the return values being the data size and a pointer pDat to a receive buffer.

Returned

0: The specified OD entry is not handled by this function.

1: The specified OD entry is handled by this function. A valid pointer and data size are returned.

4: An SDO abort SDO_ABORT_READONLY is generated.

2.6.3 The MCOUSER_AppSDOWriteComplete function

This MicroCANopen Plus function is only available when the compiler directive USECB_APPSDO_WRITE is defined. If a transfer was initiated using the function MCOUSER_AppSDOWriteInit(), then this function is called upon completion of the transfer, or if the destination buffer is full and needs to be cleared to receive more data.

Declaration

```
void MCOUSER_AppSDOWriteComplete (
  UNSIGNED8 sdoserver, // The SDO server number
  UNSIGNED16 idx, // Index of OD entry
  UNSIGNED8 subidx, // Subindex of OD entry
  UNSIGNED32 size, // number of bytes written
  UNSIGNED32 more // number of bytes still to come
  );
```

Passed

The parameters passed include the SDO server number (in range from zero to NR_OF_SDOSERVER-1), the idx (index) and subidx (subindex) of the Object Dictionary.
The value `size` indicates how many bytes were written to the receive buffer (the one specified when calling MCOUSER_AppSDOWriteInit) and `more` how many

more bytes are expected to be received. After this call data in the buffer will be overwritten starting at the beginning of the buffer.

Returned

Nothing.

2.6.4 The MCOUSER_SDORdPI function

This MicroCANopen Plus function is only available when the compiler directive USECB_SDO_RD_PI is set to one. With this function MicroCANopen Plus signals the application that an SDO Read Request was received and is now to be served. The application can use this call to either update the data or deny access.

Declaration

```
UNSIGNED32 MCOUSER_SDORdPI (
  UNSIGNED16 index,   // Index of Object Dictionary entry
  UNSIGNED8 subindex, // Subindex of Object Dictionary entry
  UNSIGNED16 offset,  // Offset to data in process image
  UNSIGNED8 len       // Length of data
  );
```

Passed

The arguments passed to the function include the Index and Subindex of the data requested as well as the location in the process image (offset) and the length of the data.

Returned

Zero if the access is granted (MicroCANopen Plus will automatically send the appropriate SDO response message). Otherwise the SDO Abort Code to return to the SDO client requesting the data.

2.6.5 The MCOUSER_SDORdAft function

This MicroCANopen Plus function is only available when the compiler directive USECB_SDO_RD_AFTER is set to one. With this function MicroCANopen Plus signals the application that an SDO Read Request completed. The application can use this call to clear the data read or to mark it as read.

Declaration

```
void MCOUSER_SDORdAft (
  UNSIGNED16 index,   // Index of Object Dictionary entry
  UNSIGNED8 subindex, // Subindex of Object Dictionary entry
  UNSIGNED16 offset,  // Offset to data in process image
  UNSIGNED8 len       // Length of data
  );
```

Passed

The arguments passed to the function include the Index and Subindex of the data requested as well as the location in the process image (offset) and the length of the data.

Returned

Nothing.

2.6.6 The MCOUSER_SDOWrPI function

This MicroCANopen Plus function is only available when the compiler directive USECB_SDO_WR_PI is set to one. With this function MicroCANopen Plus signals the application that an SDO Write Request was received and is now to be processed. The call happens BEFORE data is copied to the process image. The application can use this call to verify if the data is within expected range and can send the SDO client sending the data an ABORT if it is not..

Declaration

```
UNSIGNED32 MCOUSER_SDOWrPI (
  UNSIGNED16 index,   // Index of Object Dictionary entry
  UNSIGNED8 subindex, // Subindex of Object Dictionary entry
  UNSIGNED16 offset,  // Offset to data in process image
  UNSIGNED8 *pDat,    // Pointer to data received
  UNSIGNED8 len       // Length of data   );
```

Passed

The arguments passed to the function include the Index and Subindex of the data requested as well as the location in the process image (offset), a pointer to the data received and the length of the data.

Returned

Zero if the access is granted (MicroCANopen Plus will automatically copy the data to the Process Image and sends the appropriate SDO response message). Otherwise the SDO Abort Code to return to the SDO client sending the data.

2.6.7 The MCOUSER_SDOWrAft function

This MicroCANopen Plus function is only available when the compiler directive USECB_SDO_WR_AFTER is set to one. With this function MicroCANopen Plus signals the application that an SDO Write Request completed.

Declaration

```
void MCOUSER_SDOWrAft (
  UNSIGNED16 index,   // Index of Object Dictionary entry
  UNSIGNED8 subindex, // Subindex of Object Dictionary entry
  UNSIGNED16 offset,  // Offset to data in process image
  UNSIGNED8 len       // Length of data
  );
```

Passed

The arguments passed to the function include the Index and Subindex of the data received as well as the location in the process image (offset) and the length of the data.

Returned

Nothing.

2.6.8 The MCOUSER_SDORequest function

LEGACY FUNCTION FOR BACKWARD COMPATIBILITY, PLEASE USE MCOUSER_AppSDOReadInit() AND MCOUSER_AppSDOWriteInit() IN ALL NEW IMPLEMENTATIONS.

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.

2.6.9 The MCOUSER_Sleep function

This function is called when a message related to the sleep and wake-up mechanism is received. It is only available when the #define USE_SLEEP is set. This mechanism was first introduced in CiA447-1.

Declaration

```
void MCOUSER_Sleep (
  UNSIGNED8 mode // 0: Wakeup message received
                 // 1: Query Sleep Objection msg received
                 // 2: Set Sleep Mode message received
  );
```

Passed

The value `mode` is set to 0, 1 or 2 depending on which kind of message was received.
0 – The wakeup message was received, all nodes to wake up.
1 – The sleep objection message was received, at least one node objects to sleeping (partial power down)
2 – The set sleep mode message was received, a request to power down.
The function `MCO_TransmitSleepObjection()` should be called if the local node objects to power down.

Returned

Nothing.

2.7 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.

2.7.1 *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
  );
```

2.7.2 *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
  );
```

2.7.3 *The MCOHW_GetStatus function*

```
/*******************************************************************
DOES:    This function returns the global status variable.
CHANGES: Status can be changed anytime by this module, for example from
         within an interrupt service routine or by any of the other
         functions in this module.
BITS:    0: INIT - set to 1 after a completed initialization
                   left 0 if not yet inited or init failed
         1: CERR - set to 1 if a CAN bit or frame error occurred
         2: ERPA - set to 1 if a CAN "error passive" occurred
         3: RXOR - set to 1 if a receive queue overrun occurred
         4: TXOR - set to 1 if a transmit queue overrun occurred
         5: Reserved
         6: TXBSY - set to 1 if Transmit queue is not empty
         7: BOFF - set to 1 if a CAN "bus off" error occurred
*******************************************************************/
UNSIGNED8 MCOHW_GetStatus (
  void
  );
```

2.7.4 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 *pTransmitBuf // Data structure with message to be send
  );
```

2.7.5 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 *pTransmitBuf // Data structure with message received
  );
```

2.7.6 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 1ms tick may consider incrementing
         the timer tick only once every "x" ms, if the increment is by
         "x".
**************************************************************************/
UNSIGNED16 MCOHW_GetTime (
  void
  );
```

2.7.7 The MCOHW_IsTimeExpired function

```
/**************************************************************************
DOES:    This function compares a UNSIGNED16 timestamp to the internal
         timer tick and returns 1 if the timestamp expired/passed.
RETURNS: 1 if timestamp expired/passed
         0 if timestamp is not yet reached
NOTES:   The maximum timer runtime measurable is 0x8000 (about 32 secs).
         For the usage in MicroCANopen that is sufficient.
**************************************************************************/
UNSIGNED8 MCOHW_IsTimeExpired (
  UNSIGNED16 timestamp // Timestamp to be checked for expiration
  );
```

2.7.8 The NVOL_Init function (Plus)

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

```
/**************************************************************************
DOES:    Initializes access to non-volatile memory.
**************************************************************************/
void NVOL_Init (
  void
  );
```

2.7.9 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
  );
```

2.7.10 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
  );
```

2.7.11 The NVOL_WriteComplete function (Plus)

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

```
/**********************************************************************
DOES:    Is called when a consecutive block of write cycles is complete.
         The driver may buffer the data from calls to NVOL_WriteByte with
         consecutive destination addrs. in RAM and then write the entire
         buffer to non-volatile memory upon a call to this function.
**********************************************************************/
void NVOL_WriteComplete (
  void
  );
```

2.7.12 The MCOHWMGR_SetCANFilter function (MGR)

This function is only needed when the Manager functionality is used. CAN messages received for the Manager are received in an additional receive queue from which they are polled with an own Pull function (see below).

```
/**********************************************************************
DOES:    This function implements an additional CAN receive filter
         used by the manager. Messages received using this ID are pulled
         by the manager using function MCOHWMGR_PullMessage
         Filter set receives msgs from 0x81 to 0xFF and 0x581 to 0x5FF
RETURNS: TRUE or FALSE, if filter was not set
**********************************************************************/
UNSIGNED8 MCOHWMGR_SetCANFilter
  (
  void
  );
```

2.7.13 The MCOHWMGR_PullMessage function (MGR)

This function is only needed when the Manager functionality is used. CAN messages received for the Manager are received in an additional receive queue from which they are polled with this Pull function.

```
/**********************************************************************
DOES:    This function is used by the manager to poll messages that are
         needed by the manager
RETURNS: TRUE or FALSE, if no message was received
**********************************************************************/
UNSIGNED8 MCOHWMGR_PullMessage (
  CAN_MSG *pReceiveBuf // buffer to witch a received message is copied
  );
```

2.8 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.

2.8.1 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.

2.8.2 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.

2.8.3 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_IMGSIZE [num]

This value specifies the total size of the process image in bytes.
It must not exceed FFFEh.

3.2 GENERAL SETTINGS OF NODECFG.H

3.2.1 #define USE_MCOP [0|1]

This setting enables the "Plus" functionality of MicroCANopen Plus. It must be set to 1 when any of the functions in the module mcop.c are used.

3.2.2 #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.

3.2.3 #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.

3.3 PDO SETTINGS OF NODECFG.H

3.3.1 #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.

3.3.2 #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.

3.3.3 #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).

3.3.4 #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, MicroCANopen automatically does a divide or multiply by 10 when communicating the inhibit time via SDO requests/responses.

3.3.5 #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.

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

3.4.1 #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.

3.4.2 #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 MicroCANopen and is specified in milliseconds.

3.4.3 #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.

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

When USE_EMCY is enabled, 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. 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.

3.4.5 #define USE_NODE_GUARDING [0|1]

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 setting is enabled.

3.4.6 #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 MicroCANopen 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.

3.4.7 #define NR_OF_SDOSERVER [num]

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

3.4.8 #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.

3.5 OTHER SETTINGS OF NODECFG.H *(PLUS)*

3.5.1 #define USE_CiA447 [0|1]

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

3.5.2 #define USE_SDOMESH [0/1]

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

3.6 USER CALL-BACK FUNCTIONS OF NODECFG.H *(PLUS)*

3.6.1 #define USECB_NMTCHANGE [0|1]

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

3.6.2 #define USECB_SYNCRECEIVE [0|1]

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

3.6.3 #define USECB_RPDORECEIVE [0|1]

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

3.6.4 #define USECB_ODDATARECEIVED [0|1]

When USECB_ODDATARECEIVED is enabled, MicroCANopen 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 accesses.

3.6.5 #define USECB_TPDORDY [0/1]

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

3.6.6 #define USECB_SDOREQ [0/1]

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

3.6.7 #define USECB_SDO_RD_PI [0/1]

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

3.6.8 #define USECB_SDO_RD_AFTER [0/1]

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

3.6.9 #define USECB_SDO_WR_PI [0/1]

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

3.6.10 #define USECB_SDO_WR_AFTER [0/1]

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

3.6.11 #define USECB_APPSDO_READ [0/1]

When USECB_APPSDO_READ is enabled, MicroCANopen 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 MicroCANopen Plus of the location and size of the buffer that contains the "response".

3.6.12 #define USECB_APPSDO_WRITE [0|1]

When USECB_APPSDO_WRITE is enabled, MicroCANopen 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. MicroCANopen copies the data received to the location specified.

With MCOUSER_AppSDOWriteComplete() MicroCANopen 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.

4 SDO FULLY-MESHED COMMUNICATION

For small networks of up to 16 nodes, MicroCANopen Plus together with the manager add-on features a built-in configuration that allows any node to access all of the object dictionary of another node at any time without restrictions. This is done by enabling 16 SDO server channels and 16 SDO clients in each node and setting the channels up using a custom COB-ID assignment scheme.

4.1 PREREQUISITES

Since the fully-meshed setup uses a custom COB-ID assignment scheme for the SDO channels, all nodes have to support it. Essentially, this means that all nodes have to use MicroCANopen Plus with SDO FULLY-MESHED enabled.

Since a successful SDO communication needs both the server and the client, and the SDO client is part of the MicroCANopen Plus manager add-on, this functionality needs the manager add-on to function.

4.2 LIMITATIONS

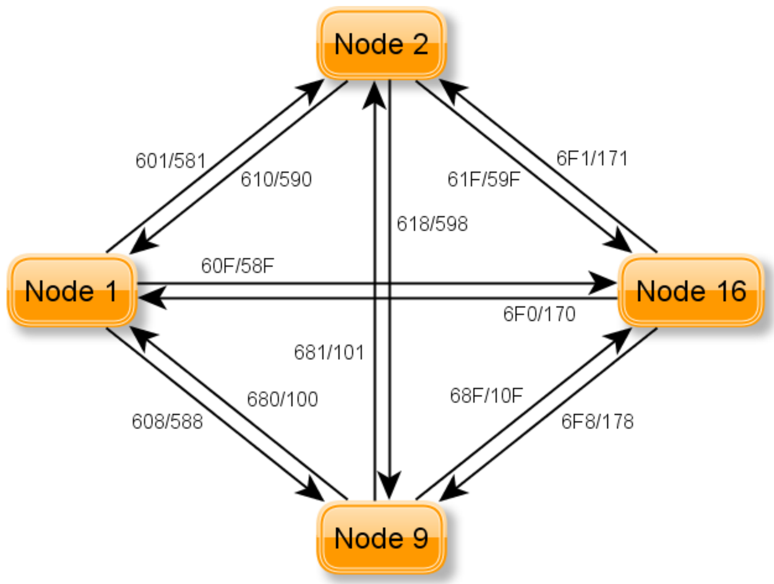
Regular CANopen nodes with default (CiA301) SDO servers cannot be combined with MicroCANopen Plus “SDO fully meshed” nodes on the same network.

The scheme cannot be used in a network where the time stamp object (COB-ID 100h) is used, when both the node IDs 9 and 1 are present and node 9 accesses node 1, using the scheme.

The scheme is not useful if nodes don't have access to the SDO client functions only present in the manager add-on module.

4.3 SDO COMMUNICATION SETUP

The following graph illustrates the SDO communication setup. The arrows point from the node sending the request (SDO client) to the node responding (SDO server). The first number accompanying each arrow is the COB-ID used for the request, the second number is the COB-ID of the response:



SDO Request COB-ID		**SDO Response COB-ID**	
First digit:	6	First digit:	5 (Client Node ID 1..8) 1 (Client Node ID 9..16)
Second digit:	Client Node ID - 1	Second digit:	Client Node ID + 7 (Client Node ID 1..8) Client Node ID - 9 (Client Node ID 9..16)
Third Digit:	Server Node ID - 1	Third Digit:	Server Node ID - 1

4.4 USAGE EXAMPLE (WITH MANAGER ADD-ON)

For the meshed communication, the stack automatically defines macros to set up the SDO client channels:

```
CAN_ID_SDOREQUEST(client,server)
CAN_ID_SDORESPONSE(client,server)
```

They should be used with client set to MY_NODE_ID. The following code illustrates a non-blocking SDO access, using call-backs.

Variables:

```
SDOCLIENT *pSC; // pointer to SDO client
UNSIGNED8 chn; // channel usage
UNSIGNED8 buf_SDO[4];
```

Setting up channel and starting request in the application:

```
chn = 0;
// Set up SDO channel 1 to talk to node 9 using buf_SDO data
pSC = MGR_InitSDOClient(1,CAN_ID_SDOREQUEST(MY_NODE_ID,9),
                        CAN_ID_SDORESPONSE(MY_NODE_ID,9),
                        &(buf_SDO[4]),4);
```

```
// Start read request of [0x1000,0x00] of remote node
MGR_SDOClientRead(pSC,0x1000,0x00);
chn = 1;   // chn is in use
```

The manager processing function `MGR_ProcessMGR()` has to be called periodically. When the request is completed, it will call `MGRCB_SDOComplete`:

```
/****************************************************************
DOES:    Called when an SDO client transfer is completed
RETURNS: nothing
****************************************************************/
void MGRCB_SDOComplete (
  UNSIGNED8 channel,    // SDO channel number in range of 1 to
                        // NR_OF_SDO_CLIENTS
  UNSIGNED32 abort_code // status, error, abort code
  )
{
  // SDO Client example of non-blocking version
  if (channel == 1)
  {
    if ((abort_code == SDOERR_READOK) ||
        (abort_code == SDOERR_WRITEOK))
    {
      chn++; // signal OK
    }
    else
    {
      chn = 0xFF; // signal ABORT
    }
  }
}
```

In the application, we may react to our SDO status variable:

```
if (chn > 1)
  if (chn = 0xFF)
    [process SDO abort]
  else
    [start next request to node 9]
```

5 APPENDIX - 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"*

5.1 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.

5.2 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.

5.2.1 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.

5.2.2 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 usage 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 hearbeat consumers must be initialized later, after MGR_InitMgr() has been called to initialize the manager.

5.2.3 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[] =
{
  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
```

6 APENDIX – ADVANCED MANUAL CONFIGURATION

6.1 RTOS INTEGRATION

The most simplistic way to integrate MicroCANopen with a Real-Time Operating System is to call *MCO_ProcessStack()* periodically, for example from a one millisecond timer task.

6.1.1 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);
```

6.1.2 PROCESS IMAGE INTEGRETY

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.

6.2 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 *entriesandreplies.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.

6.2.1 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.

6.2.2 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^{th} 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 8th byte of the process image is defined as follows:
ODENTRY(0x6200,0x01,1+ODRD+ODWR,7),

6.2.3 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. ODGENTRYP is for entries that are located in the process image and ODGENTRYC is used for entries using a pointer to any memory location (intended usage is for constant values, hence ‘C’).

The ODGENTRYP macro

ODGENTRYP(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 4th 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 8th 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"))