MicroControl

Systemhaus für Automatisierung

CANopen Bootloader Manual

Version 3.00

Document conventions

For better handling of this manual the following icons and headlines are used:

This symbol marks a paragraph containing useful information about the device operation or giving hints on configuration.

This symbol marks a paragraph which describes actions to be executed by the user of the source code package.

This symbol marks a paragraph which explains possible danger. This danger might cause a damage to the system or damage to personnel. Read these sections carefully!

Keywords

Important keywords appear in the border column to help the reader when browsing through this document.

Syntax, Examples

`For function syntax and code examples the font face Source Code Pro is used.`

MicroControl GmbH & Co. KG
Junkersring 23
D-53844 Troisdorf
Fon: +49 / 2241 / 25 65 9 - 0
Fax: +49 / 2241 / 25 65 9 - 11
http://www.microcontrol.net

1. Scope 5
1.1 References 6
1.2 Abbreviations 6
1.3 Definitions 7
1.4 Introduction to CAN 7
1.5 Introduction to CANopen 8

2. CANopen Bootloader Overview 9
2.1 Naming Conventions 9
2.2 Message Router 10
2.3 File Structure 11
2.4 Configuration Options 12
2.5 Initialisation Process 13

3. CANopen Bootloader Manager. 15
3.1 Initialisation 15
3.2 Manager Configuration Options 17
3.3 Manager Functions 18

4. Network Management - NMT 25
4.1 NMT Functions 25

5. Service Data Objects - SDO 29
5.1 SDO Server Configuration Options 29
5.2 SDO Server Functions 31
5.3 Object Dictionary 31

6. Emergency Service - EMCY 37
6.1 EMCY Producer Configuration Options 37
6.2 EMCY Producer Functions 38

7. Layer Setting Services - LSS. 39
7.1 LSS Configuration Options 39

8. Flash Programming 41
8.1 Erase Flash Memory 42
8.2 Store Data to Flash Memory 43
8.3 Finalize Flash Programming 45

A Source Code Agreement 47

B. Index 51

1. Scope

The CANopen Bootloader manual describes the Application Programming Interface (API) for access to the CANopen services. The API provides functionality for the CANopen standards CiA 301, CiA 302 and CiA 305.

The CANopen Bootloader Protocol Stack is independent from the used CAN hardware and operating system. Access to the CAN hardware is done via the CANpie API, which is available for a wide range of CAN controllers. The CANpie API is not subject of this manual.

1

1.1 References

/1/ CiA 301, CANopen Application Layer and Communication Profile, Version 4.2, CAN in Automation (CiA) e.V., http://www.can-cia.org

/2/ CiA 302, Additional application layer functions, Part 1 - 7, Version 4.0.2, CAN in Automation (CiA) e.V. http://www.can-cia.org

/3/ CiA 305, CANopen Layer setting services and protocols, Version 2.2, CAN in Automation (CiA) e.V., http://www.can-cia.org

/4/ CANpie users guide, Version 2.0, MicroControl GmbH & Co. KG http://www.microcontrol.net/en/source-code/canpie.html

1.2 Abbreviations

API	Application Programming Interface
CAN	Controller area network
CAN-ID	CAN identifier
COB	Communication object
COB-ID	Communication object identifier
CRC	Cyclic redundancy check
LSB	Least significant bit/byte
MSB	Most significant bit/byte
OSI	Open systems interconnection
RTR	Remote transmission request

1.3 Definitions

CAN base frame

message that contains up to 8 byte and is identified by 11 bits as defined in ISO 11898-1

CAN extended frame

message that contains up to 8 byte and is identified by 29 bits as defined in ISO 11898-1

CAN-ID

identifier for CAN data and remote frames as defined in ISO 11898-1

1.4 Introduction to CAN

CAN (Controller Area Network) is an international standard defined in the ISO 11898 for high speed and ISO 11519-2 for low speed.

CAN is based on a broadcast communication mechanism. This broadcast communication is achieved by using a message oriented transmission protocol. These messages are identified by using a message identifier. Such a message identifier has to be unique within the whole network and it defines not only the content but also the priority of the message.

The priority at which a message is transmitted compared to another less urgent message is specified by the identifier of each message. The priorities are laid down during system design in the form of corresponding binary values and cannot be changed dynamically. The identifier with the lowest binary number has the highest priority. Bus access conflicts are resolved by bit-wise arbitration on the identifiers involved by each node observing the bus level bit for bit. This happens in accordance with the "wired and" mechanism, by which the dominant state overwrites the recessive state. The competition for bus allocation is lost by all nodes with recessive transmission and dominant observation. All the "losers" automatically become receivers of the message with the highest priority and do not re-attempt transmission until the bus is available again.

The CAN protocol supports two message frame formats, the only essential difference being in the length of the identifier. The CAN standard frame, also known as CAN 2.0 A, supports a length of 11 bits for the identifier, and the CAN extended frame, also known as CAN 2.0 B, supports a length of 29 bits for the identifier.

1

1.5 Introduction to CANopen

CANopen is a communication protocol and device profile specification for embedded systems used in automation. In terms of the OSI model, CANopen implements the layers above and including the network layer. The CANopen standard consists of an addressing scheme, several small communication protocols and an application layer defined by a device profile. The communication protocols have support for network management, device monitoring and communication between nodes, including a simple transport layer for message segmentation/desegmentation. The lower level protocol implementing the data link and physical layers is usually Controller Area Network (CAN), although devices using some other means of communication (such as Ethernet Powerlink, EtherCAT) can also implement the CANopen device profile.

The basic CANopen device and communication profiles are given in the CiA 301 specification released by CAN in Automation /1/. Profiles for more specialized devices are built on top of this basic profile, and are specified in numerous other standards released by CAN in Automation, such as CiA 401 for I/O-modules and CiA 402 for motion control.

2. CANopen Bootloader Overview

The following figure shows an overview of the CANopen Bootloader functionality. Each CANopen service is described in a separate chapter.

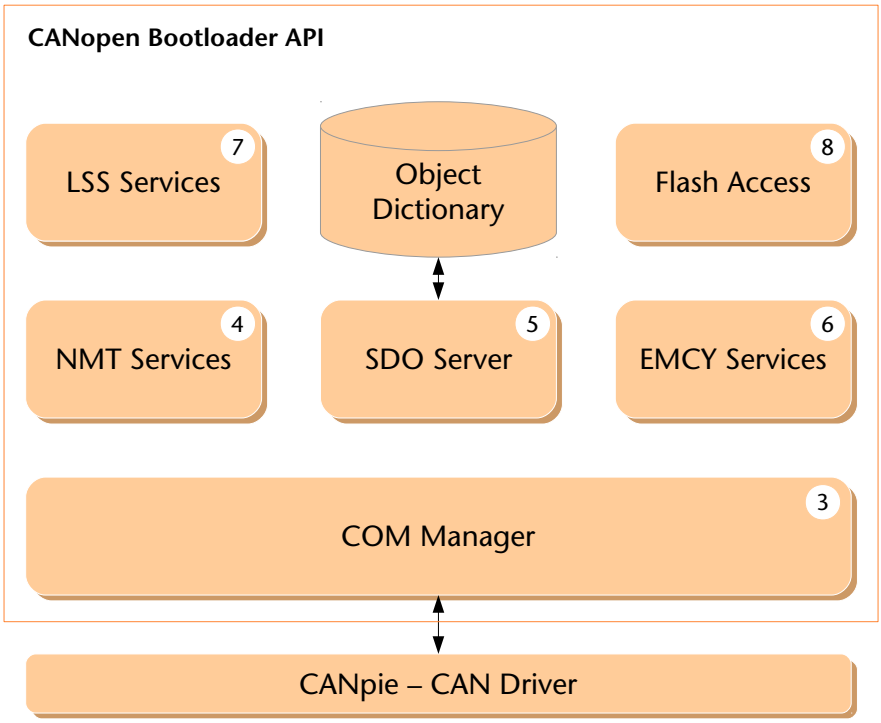

Fig. 1: CANopen Bootloader overview

The CANopen Bootloader Protocol Stack uses a well-defined CAN API (CANpie) to the CAN interface and thus can be adopted to any kind of CAN controller. The CANpie API is not described in this manual, for more information refer to /4/.

2.1 Naming Conventions

All functions, structures, defines and constant value of the CANopen Bootloader stack have the prefix "**Cbl**". The following table shows the used nomenclature:

CANopen Bootloader stack	*Prefix*
Functions	Cbl<service><name>
Enumeration	eCBL_<name>
Structures	Cbl<name>_s
Defines	CBL_<service>_<name>
Error Codes	eCBL_ERR_<name>

Table 1: Naming conventions

2.2 Message Router

The message router is responsible for reading and writing CAN messages between the CANopen Bootloader protocol stack and the CAN bus. The CANpie API /4/ and its buffer concept is used to access the CAN interface on the different target platforms.

CAN messages are transmitted and received by different CAN message buffers. Depending on the CANopen service, a specific CAN message buffer will be selected.

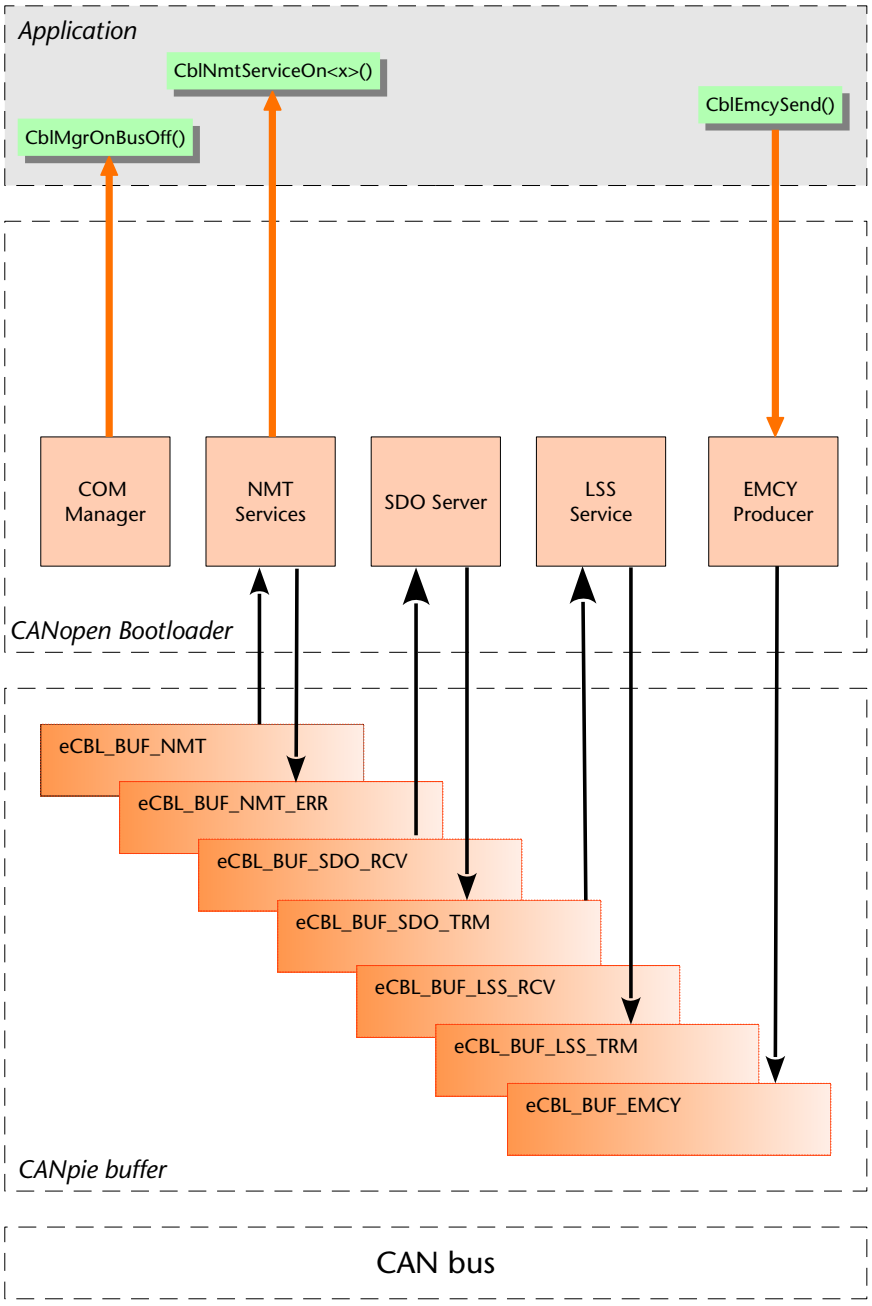

Fig. 2: Detail view of CANpie buffers and associated CANopen services

2.3 File Structure

All header files and implementation files of the CANopen Bootloader protocol stack package are located in the *source* directory.

File	*Function*
cbl_conf.h	CANopen Bootloader configuration file
cbl_dict.c / h	Object dictionary
cbl_emcy.c / h	Emergency service
cbl_led.c / h	LED support
cbl_lss.c / h	Layer Settings Service (LSS)
cbl_mgr.c / h	CANopen Bootloader manager
cbl_nmt.c / h	Network management (NMT)
cbl_objs.c / h	Object dictionary
cbl_sdo.c / h	Service data objects (SDO) server
cbl_time.c / h	Timer services
cbl_user.c	Application functions / event handler

Table 2: CANopen Bootloader protocol stack files

The file **`cbl_user.c`** contains all variables and functions that require an adoption to the target system.

2.4 Configuration Options

The file ***cbl_conf.h*** contains all definitions for the configuration of the CANopen Bootloader protocol stack. Please set the symbols to an appropriate value in order to achieve a specific CANopen Bootloader stack behaviour. The most important options are listed here. For a detailed specification please refer to the HTML documentation.

2.4.1 CBL_TIMER_PERIOD

This symbol defines the period of the timer interrupt. The value is a multiple of 1 microsecond. It is used for timing services. Please set this value to the timer interrupt period of the target system.

Please note that the value must be at least 1000 [microseconds], because all CANopen services use a multiple of 1 millisecond.

2.4.2 CBL_DICT_MAN

This symbol defines if manufacturer specific objects are included in the dictionary. A value of 0 means the manufacturer objects are not included. A value of 1 means they are included.

2.4.3 CBL_DICT_SEARCH_FAST

This symbol defines if a fast search algorithm is used for the dictionary. The fast search algorithm increases the code size. A value of 0 means a linear search method is used. A value of 1 means the fast search algorithm is used.

2.4.4 CBL_PROG_MAX

The symbol defines the number of programs (or data) that are supported by the CANopen bootloader. The value has impact on the highest supported sub-index for the objects $1F50_h$ .. $1F57_h$.

2.5 Initialisation Process

The CANopen Bootloader is initialised with **`CblMgrInit()`**. This routine will setup the CAN controller and initialise all necessary services. The support of services and their behaviour can be configured using the **`cbl_conf.h`** file. Afterwards the stack can be started by calling the **`CblMgrStart()`** to start running the CANopen Bootloader.

In summary three steps are necessary to run the CANopen Bootloader:
- Initialise CANopen Bootloader Stack
- Start CANopen Bootloader
- Start the timer event function

```
void MyCblInit(void)
{
   //------------------------------------------------------
   // Initialize the CANopen Bootloader stack
   //
   CblMgrInit(CP_CHANNEL_1, 0);

   //------------------------------------------------------
   // Start the CANopen Bootloader,
   // bitrate = 500 kBit/s, node-ID = 1,
   //
   CblMgrStart(CP_BAUD_500K, 1);

   //------------------------------------------------------
   // now the CANopen Bootloader Stack is initialized and
   // has to be triggered by calling CblMgrTimerEvent() with
   // a cycle time of CBL_TIMER_PERIOD

}
```

Example 1: Initialization process

The initialisation functions of the CANopen Bootloader protocol stack have to be executed prior to any other API functions.

2

3. CANopen Bootloader Manager

The CANopen Bootloader Manager covers the initialization and control of the protocol stack. It also manages the initialization of the CAN interface via the CANpie driver.

3.1 Initialisation

The CANopen Bootloader stack is initialized by calling the two functions **`CblMgrInit()`** and **`CblMgrStart()`**.


```
//-------------------------------------------------------
// Initialize the CANopen Bootloader stack
//
CblMgrInit(CP_CHANNEL_1, 0);

//-------------------------------------------------------
// initialization for timer and other peripheral
// can be done here


//-------------------------------------------------------
// Start the CANopen Bootloader,
// bitrate = 500 kBit/s, node-ID = 1,
//
CblMgrStart(CP_BAUD_500K, 1);
```

Example 2: Initialization of CANopen Bootloader protocol stack

After calling the **`CblMgrStart()`** CANopen Bootloader stack is running and a Bootup message is send on the CAN bus.

The next example shows a complete generic initialisation of the protocol inside the main function. Additional functions for the microcontroller and timer are provided by the MicroControl Library (MCL), they are shown for a better understanding of the example code.

3

```
//-------------------------------------------------------
// Initialize the target CPU
//
McCpuInit();

//-------------------------------------------------------
// Initialize flash
//
McFlashInit();

//-------------------------------------------------------
// Initialize the CANopen Bootloader
//
CblMgrInit(CP_CHANNEL_1, 0);

//-------------------------------------------------------
// Try to jump to application.    //
if (McIapIsAppValid() == eIAP_APP_VALID)
{
   if (McIapIsBootLocked() == eIAP_BOOT_UNLOCKED)
   {
      McIapJumpToApp();
   }
}
//-------------------------------------------------------
// Initialize the timer resource on the target CPU
//
McTmrInit();
McTmrFunctionInit(CblMgrTimerEvent,
                  McTmrTimeToTicks(1000),
                  eTMR_CTRL_START);
//-------------------------------------------------------
// Start the CANopen Bootloader,
// bitrate = 500 kBit/s, node-ID = 1,
//
CblMgrStart(CP_BAUD_500K, 1);
CblLedNetworkStatus(eCblLedNet_PREOPERATIONAL);

//-------------------------------------------------------
// this is the main loop of the embedded application
//
while (1)
{
   //----------------------------------------------------
   // check the result of the CANopen manager call
   //
   if(CblMgrProcess() != eCblErr_NODE_RESET)
   {

   }
   else
   {
      CblMgrRelease();
  }
}  // end while (1)
```

Example 3: Complete generic initialization of CANopen Bootloader stack

3.2 Manager Configuration Options

The file ***cbl_conf.h*** contains definitions for the configuration of the Manager module. Please set the symbols to an appropriate value in order to achieve a specific CANopen Manager behaviour. For a detailed specification please refer to the HTML documentation.

3.2.1 CBL_MGR_INT

With this symbol it is possible to switch the CAN message handler (message reception) between Polling- and IRQ-mode. In Polling mode the messages are read from the buffer during the main loop. The default mode is the IRQ-mode: received CAN messages are processed inside the CAN IRQ-handler.

Prior to changing this symbol make sure that the CANpie driver supports the requested method.

3.2.2 CBL_TMR_INT

With this symbol it is possible to switch the timer function between Polling- and IRQ-mode. In Polling mode the timer value is checked within the main loop. The default mode is the IRQ-mode: the function `CblMgrTimerEvent()` is called within the timer interrupt.

3.3 Manager Functions

The manager functions (prefix **CblMgr**) provide general control over the CANopen Bootloader protocol stack.

Function name	Description
CblMgrBufferCrc()	Calculate checksum of memory
CblMgrBufferStore()	Store program / data Callback function (cbl_user.c)
CblMgrEventBusOff()	CAN bus status change to Bus-Off, Callback function (cbl_user.c)
CblMgrEventProgControl()	Handle manufacturer specific program control Callback function (cbl_user.c)
CblMgrFlashDelete()	Delete program / data in Flash memory Callback function (cbl_user.c)
CblMgrFlashFinalize()	Finalize program / data in Flash memory Callback function (cbl_user.c)
CblMgrInit()	Initialise protocol stack and CAN interface
CblMgrRelease()	Shutdown protocol stack and CAN interface
CblMgrStart()	Start CANopen Bootloader
CblMgrTimerEvent()	Handler for periodic services

Table 3: Functions of CANopen Bootloader Manager

3.3.1 CblMgrBufferCrc

Syntax

```
uint32_t CblMgrBufferCrc(
    uint8_t *       pubMemStartV,
    uint32_t        ulSizeV)
```

Function

Calculate buffer CRC

This function calculates the CRC-32 value of the memory starting at **pubMemStartV** with a size of **ulSizeV** bytes. The algorithm is explained at http://de.wikipedia.org/wiki/CRC32.

Parameters

pubMemStartV Pointer to beginning of memory

ulSizeV Number of bytes

Return value

CRC32 value

3.3.2 CblMgrBufferStore

Syntax

```
void CblMgrBufferStore(
    uint8_t         ubProgNumV,
    uint8_t *       pubBufferV,
    uint16_t        uwSizeV)
```

Function

Store program / data

This function stores the transferred data from the RAM buffer into the Flash memory. The parameter **ubProgNumV** defines the program number and is equivalent to the entries of object $1F50_h$ (see "Program Data" on page 35). The pointer **pubBufferV** denotes the beginning of the RAM buffer. The parameter **uwSizeV** defines the number of bytes to be stored.

User

Since the implementation depends on the used microcontroller and application, this function has to be adopted to the hardware. Please refer to "Store Data to Flash Memory" on page 43 for a sample implementation.

Parameters

ubProgNumV Program number

pubBufferV Pointer to source data

uwSizeV Number of bytes to copy to Flash memory

Return value

None

3.3.3 CblMgrEventBusOff

Syntax

```
void CblMgrEventBusOff(void)
```

Function

Handle Bus-Off condition

User

This function handles a bus-off condition. Since the behaviour of this function is application specific, the implementation is available in the file **cbl_user.c**.

3

Parameters

None

Return value

None

3.3.4 CblMgrEventProgControl

Syntax

```
void CblMgrEventProgControl(
    uint8_t         ubProgNumV,
    uint8_t         ubProgControl)
```

Function

Manufacturer-specific program control

User

This function is called upon writing a value greater or equal 80_h to object $1F51_h$ (program control). Since the behaviour of this function is application specific, the implementation is available in the file **cbl_user.c**.

Parameters

ubProgNumV	Program number
ubProgControlV	Program control value written to $1F51_h$

Return value

None

3.3.5 CblMgrFlashDelete

Syntax

```
void CblMgrFlashDelete(
    uint8_t         ubProgNumV)
```

Function

Delete program / data in Flash memory

This function deletes the program with number **ubProgNumV** inside the Flash memory. The parameter **ubProgNumV** defines the program number and is equivalent to the entries of object $1F50_h$ (see "Program Data" on page 35).

User

Since the implementation depends on the used microcontroller and application, this function has to be adopted to the hardware. Please refer to "Erase Flash Memory" on page 42 for a sample implementation.

Parameters

ubProgNumV	Program number

Return value

None

3.3.6 CblMgrFlashFinalize

Syntax

```
void CblMgrFlashFinalize(
    uint8_t         ubProgNumV,
    uint32_t        ulSizeV)
```

Function

Finalize program / data in Flash memory

This function finalizes the program with number **ubProgNumV** inside the Flash memory. The parameter **ubProgNumV** defines the program number and is equivalent to the entries of object $1F50_h$ (see "Program Data" on page 35). The parameter **ulSizeV** denotes the number of bytes that have been transferred during the SDO download procedure.

User

Since the implementation depends on the used microcontroller and application, this function has to be adopted to the hardware. Please refer to "Finalize Flash Programming" on page 45 for a sample implementation.

Parameters

ubProgNumV	Program number
ulSizeV	Program size

Return value

None

3.3.7 CblMgrInit

Syntax

```
uint8_t CblMgrInit(
    uint8_t         ubCanIfV,
    uint16_t        uwConfigV)
```

Function

Initialize CANopen Bootloader stack

This function initialises the CANopen Bootloader stack and must be called prior to any other function of the CANopen Bootloader stack. It is responsible for the initialisation of all services (NMT, SDO, etc.).

The function assigns the CANopen Bootloader stack to the CAN interface given by **ubCanIfV**.

The parameter **uwConfigV** is not evaluated by the function.

The usage of this function is shown by an example in "Initialisation" on page 15.

Parameters

ubCanIfV	CAN interface index

Return value

On success the value **eCblErr_OK** is returned.

3.3.8 CblMgrRelease

Syntax

```
uint8_t CblMgrRelease(void)
```

Function

Shutdown the CANopen Bootloader

This function performs a shutdown of the CANopen Bootloader.

Parameters

None

Return value

On success the value **eCblErr_OK** is returned.

3.3.9 CblMgrStart

Syntax

```
int8_t CblMgrStart(
    uint8_t         ubBaudSelV,
    uint8_t         ubNodeIdV)
```

Function

Start the CANopen Bootloader

This functions starts the CANopen Bootloader protocol stack. A boot-up message (ID = 700_h + node-ID) is generated on the CAN bus.

Parameters

ubBaudSelV Initial bit-rate

ubNodeIdV Device node-ID

Return value

On success the value **eCblErr_OK** is returned.

3.3.10 CblMgrTimerEvent

Syntax `void CblMgrTimerEvent(void)`

Function Execute Timer-based Services

This function must be called periodically by a timer resource of the target system. It is responsible to call CANopen services that depend on a timer (e.g. Heartbeat).

3

Parameters None

Return value None

```
//-------------------------------------------------------------//
// Timer interrupt service routine                             //
//                                                             //
//-------------------------------------------------------------//
void MyTimerInterrupt(void)
{
   //... timer services of application ...
   //--- call CANopen stack timer function -------------
   CblMgrTimerEvent();

   //... retrigger the timer
}
```

Example 4: Example routine for CblTmrEvent()

In order to have periodical functions available (e.g. heartbeat), it is necessary to call the function **`CblTmrEvent()`** cyclically. The cycle time is defined in microseconds by **`CBL_TIMER_PERIOD`** in **`cbl_conf.h`** and must match the trigger time.

Typically **`CblTmrEvent()`** will be called from a timer interrupt but it's also possible to call it from main loop. This behaviour is controlled by **`CBL_TMR_INT`** defined in **`cbl_conf.h`**.

4. Network Management - NMT

The network management (NMT) is CANopen device oriented and follows a master-slave structure. NMT objects are used to execute NMT services. Through NMT services CANopen devices are initialised, started, monitored, reset or stopped.

All CANopen devices are regarded as NMT slaves. An NMT slave is uniquely identified in the network by its node-ID, a value in the range of 1 to 127. NMT requires that one CANopen device in the network fulfils the function of the NMT master /1/.

4.1 NMT Functions

The Network Management functions of the CANopen Bootloader protocol stack have the prefix **`CblNmt`**.

Function name	Description
`CblNmtEventStart`	Handler for NMT Start, Callback function (cbl_user.c)
`CblNmtEventStop`	Handler for NMT Stop, Callback function (cbl_user.c)
`CblNmtEventPreOperational`	Handler for NMT Pre-Operational, Callback function (cbl_user.c)
`CblNmtSetHeartbeatProd`	Configure heartbeat producer

Table 4: NMT functions

4.1.1 CblNmtEventStart

Syntax `void CblNmtEventStart(void)`

Function This service routine is called when the NMT state machine changes into "NMT Operational" state. It can be used to perform device specific routines after reception of the NMT command.

Since the behaviour of this function is application specific, the implementation is available in the file **`cbl_user.c`**.

Parameters None

Return value None

4.1.2 CblNmtEventStop

Syntax `void CblNmtEventStop(void)`

Function This service routine is called when the NMT state machine changes into "NMT Stopped" state. It can be used to perform device specific routines after reception of the NMT command.

Since the behaviour of this function is application specific, the implementation is available in the file **`cbl_user.c`**.

Parameters None

Return value None

4.1.3 CblNmtEventPreOperational

Syntax

```
void CblNmtEventPreOperational(void)
```

Function

This service routine is called when the NMT state machine changes into "NMT Pre-Operational" state. It can be used to perform device specific routines after reception of the NMT command.

Since the behaviour of this function is application specific, the implementation is available in the file **cbl_user.c**.

Parameters

None

Return value

None

4.1.4 CblNmtSetHeartbeatProd

Syntax

```
void CblNmtSetHeartbeatProd(
    uint16_t        uwTimeV)
```

Function

This function sets the heartbeat producer time (Object 1017_h). The parameter **uwTimeV** denotes the time in milli-seconds. By default, the heartbeat producer time is set to 0 during initialization.

Parameters

uwTimeV	Heartbeat cycle time

Return value

None

4

5. Service Data Objects - SDO

With Service Data Objects (SDOs) the access to entries of a device Object Dictionary is provided. As these entries may contain data of arbitrary size and data type SDOs can be used to transfer multiple data sets (each containing an arbitrary large block of data) from a client to a server and vice versa.

The client can control via a multiplexor (index and sub-index of the Object Dictionary) which data set is to be transferred. The contents of the data set are defined within the Object Dictionary /1/.

5.1 SDO Server Configuration Options

The file ***cbl_conf.h*** holds definitions for the configuration of the SDO server module. Please set the symbols to an appropriate value in order to achieve a specific SDO server behaviour. For a detailed specification please refer to the HTML documentation.

5.1.1 CBL_SDO_SEGMENTED_RD

The symbol defines if the segmented SDO Read transfer is supported. If segmented SDOs are not supported, the code size can be reduced. However, segmented SDOs are required if the data type "Visible string" (`CoDT_VISIBLE_STRING`) need to be supported with string length greater than 4.

5.1.2 CBL_SDO_SEGMENTED_WR

The symbol defines if the segmented SDO Write transfer is supported. If segmented SDOs are not supported, the code size can be reduced. However, segmented SDO write is required if the download tool only supports this SDO protocol type.

5.1.3 CBL_SDO_BLOCK

The symbol defines if the SDO Block transfer is supported. A value of 0 denotes that SDO Block transfer is not supported. A value greater 0 denotes the maximum number of blocks that can be transferred between master and slave. The maximum value is 127 blocks.

5.1.4 CBL_SDO_BLOCK_RD

The symbol defines if the Block SDO Read transfer is supported. If Block read is not supported, the code size can be reduced.

5.1.5 CBL_SDO_BLOCK_WR

The symbol defines if the Block SDO Write transfer is supported. If Block write is not supported, the code size can be reduced. However, Block SDO write is required if the download tool only supports this SDO protocol type.

5.1.6 CBL_SDO_ABORT_CODE

The symbol defines if the complete range of SDO Abort Codes is supported. A limited number of SDO Abort Codes will generate smaller code.

5.2 SDO Server Functions

The SDO server functions of the CANopen Bootloader protocol stack have the prefix **CblSdo** and are located in the file **cbl_sdo.c** within the source directory.

5.3 Object Dictionary

The following chapter describes the implementation details of all supported objects.

Index	Name	Configuration
1000_h	Device Profile	-
1001_h	Error Register	-
1002_h	Manufacturer Status	-
1008_h	Manufacturer Device Name	CBL_DICT_OBJ_1008
1009_h	Manufacturer Hardware Version	CBL_DICT_OBJ_1009
$100A_h$	Manufacturer Software Version	CBL_DICT_OBJ_100A
1014_h	COB-ID Emergency	CBL_DICT_OBJ_1014
1017_h	Heartbeat Producer Time	-
1018_h	Identity Object	-
$1F50_h$	Program Data	CBL_PROG_MAX
$1F51_h$	Program Control	CBL_PROG_MAX
$1F56_h$	Application software identification	CBL_PROG_MAX
$1F57_h$	Flash status identification	CBL_PROG_MAX

Table 5: Supported objects for bootloader

The application can access the object values via global variables. Global variables have been chosen (in contrast to static variables with Getter / Setter functions) in order to keep the memory requirements low. The variables are explained inside the object description.

Device Profile

Index 1000_h

The object at index 1000_h describes the type of device and its functionality.

Sub-Index	Data Type	Acc.	Name	Default Value
0	Unsigned32	ro	Device Profile	0000 0000h

The object is read-only. Only sub-index 0 is supported. An access to other sub-indices will lead to an abort message.

The variable **`ulIdx1000_DeviceTypeG`** defines the value of this object. The default value is assigned in the **`cbl_user.c`** file.

Error Register

Index 1001_h

The object at index 1001_h is the error register for the device.

Sub-Index	Data Type	Acc.	Name	Default Value
0	Unsigned8	ro	Error Register	-

The object is read-only. Only sub-index 0 is supported. An access to other sub-indices will lead to an abort message.

The variable **`ubIdx1001_ErrorRegisterG`** defines the value of this object. The default value is assigned during initialization of the protocol stack.

Manufacturer Status Register

Index 1002_h

The object at index 1002_h is the manufacturer status register for the device.

Sub-Index	Data Type	Acc.	Name	Default Value
0	Unsigned32	ro	Man, Status Register	$0000\ 0000_h$

The object is read-only. Only sub-index 0 is supported. An access to other sub-indices will lead to an abort message.

The variable **`ulIdx1002_StatusRegisterG`** defines the value of this object. The default value is assigned during initialization of the protocol stack.

Manufacturer Device Name

Index 1008_h

The object at index 1008_h contains the manufacturer device name.

Sub-Index	Data Type	Acc.	Name	Default Value
0	Visible String	ro	Device name	Boot

The object is read-only. Only sub-index 0 is supported. An access to other sub-indices will lead to an abort message.

Support of this object can be enabled or disabled via the configuration symbol **`CBL_DICT_OBJ_1008`**.

The variable **`ubIdx1008_DeviceNameG[]`** defines the value of this object. The default value is assigned in the **`cbl_user.c`** file.

Manufacturer Hardware Version

Index 1009_h

The object at index 1009_h contains the manufacturer hardware version.

Sub-Index	Data Type	Acc.	Name	Default Value
0	Visible String	ro	Hardware version	1.00

The object is read-only. Only sub-index 0 is supported. An access to other sub-indices will lead to an abort message.

Support of this object can be enabled or disabled via the configuration symbol **`CBL_DICT_OBJ_1009`**.

The variable **`ubIdx1009_HwVersionG[]`** defines the value of this object. The default value is assigned in the **`cbl_user.c`** file.

Manufacturer Software Version

Index $100A_h$

The object at index $100A_h$ contains the manufacturer software version.

Sub-Index	Data Type	Acc.	Name	Default Value
0	Visible String	ro	Software version	1.00

The object is read-only. Only sub-index 0 is supported. An access to other sub-indices will lead to an error message.

Support of this object can be enabled or disabled via the configuration symbol **`CBL_DICT_OBJ_100A`**.

The variable **`ubIdx100A_SwVersionG[]`** defines the value of this object. The default value is assigned in the **`cbl_user.c`** file.

Producer Heartbeat Time

Index 1017_h

The object at index 1017_h defines the cycle time of the heartbeat. The producer heartbeat time is 0 if it is not used. The time is a multiple of 1ms.

Sub-Index	Data Type	Acc.	Name	Default Value
0	Unsigned16	rw	Producer Time	0

Only sub-index 0 is supported. An access to other sub-indices will lead to an error message.

The application can modify the value of this object by the function `CblNmtSetHeartbeatProd()`.

Identity Object

5

Index 1018_h

The object at index 1018_h allows identification of the device.

Sub-Index	Data Type	Acc.	Name	Default Value
0	Unsigned8	ro	Highest sub-index	4
1	Unsigned32	ro	Vendor ID	-
2	Unsigned32	ro	Product Code	-
3	Unsigned32	ro	Revision Number	-
4	Unsigned32	ro	Serial Number	-

The object is read-only. Only sub-indices 0 to 4 are supported. An access to other sub-indices will lead to an error message.

The following variables define the value of these object:

Object	Variable / Function
$1018_h:01_h$	`ulIdx1018_VendorIdG`
$1018_h:02_h$	`ulIdx1018_ProductCodeG`
$1018_h:04_h$	`ulIdx1018_RevisionNumG`
$1018_h:04_h$	`uint32_t CblMgrGetSerialNumber()`

Table 6: Setting default values for Identity Object

The default values are assigned in the **`cbl_user.c`** file.

Program Data

Index $1F50_h$

The object at index $1F50_h$ is used to download a new application firmware. The CANopen device accepts a download request to $1F50_h$ only, if its program status is stopped (refer to $1F51_h$). With completion of the SDO transfer, it is responded and the object $1F57_h$ will be set accordingly.

Sub-Index	Data Type	Acc.	Name	Default Value
0	Unsigned8	ro	Highest sub-index	1
1	Domain	wo	Program Data 1	-
..	..	..	..	..
N	Domain	wo	Program Data N	-

Only sub-indices 0 to CBL_PROG_MAX are supported. An access to other sub-indices will lead to an error message.

The number of sub-indices (i.e. the number of different programs) is controlled via the configuration symbol CBL_PROG_MAX. The SDO protocol type for the download (segmented / block) is controlled via the configuration symbols described in refer to “SDO Server Configuration Options” on page 29.

Program Control

Index $1F51_h$

The object at index $1F51_h$ is used for the control of the programs downloaded to the CANopen device.

Sub-Index	Data Type	Acc.	Name	Default Value
0	Unsigned8	ro	Highest sub-index	1
1	Unsigned8	rw	Program Control 1	00_h
..	..	..	..	..
N	Unsigned8	rw	Program Control N	00_h

Only sub-indices 0 to CBL_PROG_MAX are supported. An access to other sub-indices will lead to an error message.

The number of sub-indices (i.e. the number of different programs) is controlled via the configuration symbol CBL_PROG_MAX.

The variable **aubIdx1F51_ControlG[]** defines the value of this object. The default value (**eCBL_PROG_CONTROL_STOP**) is assigned during initialization of the protocol stack.

Program Software Identification

Index $1F56_h$

If a CANopen device supports program software download, a network configuration tool or a CANopen manager may use this object to verify the version of the program software.

Sub-Index	Data Type	Acc.	Name	Default Value
0	Unsigned8	ro	Highest sub-index	1
1	Unsigned32	ro	Program Number 1	-
..	..	..	..	..
N	Unsigned32	ro	Program Number N	-

Only sub-indices 0 to `CBL_PROG_MAX` are supported. An access to other sub-indices will lead to an error message.

The number of sub-indices (i.e. the number of different programs) is controlled via the configuration symbol `CBL_PROG_MAX`.

5

The variable **`aulIdx1F56_SoftIdentG[]`** defines the value of this object. The function `CblMgrBufferCrc()` may be used to calculate the value of this object. The object value is typically assigned inside the `CblMgrFlashFinalize()` function call, which is application specific. Other functions for CRC calculation may be used and can be added to `CblMgrFlashFinalize()`.

Flash status identification

Index $1F57_h$

The object at index $1F57_h$ is used to display the current flash memory status.

Sub-Index	Data Type	Acc.	Name	Default Value
0	Unsigned8	ro	Highest sub-index	1
1	Unsigned32	ro	Program Number 1	$0000\ 0000_h$
..	..	..	..	..
N	Unsigned32	ro	Program Number N	$0000\ 0000_h$

Only sub-indices 0 to `CBL_PROG_MAX` are supported. An access to other sub-indices will lead to an error message.

The number of sub-indices (i.e. the number of different programs) is controlled via the configuration symbol `CBL_PROG_MAX`.

The variable **`aulIdx1F57_FlashStatusG[]`** defines the value of this object.

6. Emergency Service - EMCY

Emergency objects are triggered by the occurrence of a device internal error situation and are transmitted from an emergency producer on the device. Emergency objects are suitable for interrupt type error alerts.

An emergency object is transmitted only once per 'error event'. As long as no new errors occur on a device, no further emergency objects must be transmitted.

The emergency object may be received by zero or more emergency consumers. The reaction on the emergency consumer(s) is not specified /1/.

6.1 EMCY Producer Configuration Options

The file ***cbl_conf.h*** holds definitions for the configuration of the EMCY producer service. Please set the symbols to an appropriate value in order to achieve a specific EMCY behaviour. For a detailed specification please refer to the HTML documentation.

6.1.1 CBL_DICT_OBJ_1014

This symbol defines if the object 1014_h (EMCY object) and also the EMCY service (CblEmcySend()) is supported by the device.

6.2 EMCY Producer Functions

The EMCY producer functions of the CANopen Bootloader protocol stack have the prefix **CblEmcy** and are located in the file **cbl_emcy.c** within the source directory.

Function name	Description
CblEmcySend()	Send an EMCY message

Table 7: EMCY Producer Service functions

6.2.1 CblEmcySend

Syntax

```
void CblEmcySend(
    uint16_t        uwEmcyCodeV,
    uint8_t *       pubManCodeV)
```

Function

This function is used to send an emergency message (EMCY). The possible values for **uwEmcyCodeV** are defined in CiA 301. The parameter **pubManCodeV** is a pointer to an array of 5 bytes, which can be filled with user defined data.

6

Parameters

uwEmcyCodeV Emergency code

pubManCodeV Pointer to manufacturer data

Return value

None

```
uint8_t aubDataT[5];

aubDataT[0] = 0x01;
aubDataT[1] = 0x02;
aubDataT[2] = 0x03;
aubDataT[3] = 0x04;
aubDataT[4] = 0x05;

CblEmcySend(EMCY_ERR_DEV_GENERAL, &aubDataT[0]);
```

Example 5: Transmission of EMCY message

7. Layer Setting Services - LSS

LSS offers the possibility to inquire and change the settings of certain parameters of the local layers on a CANopen module with LSS Slave capabilities by a CANopen module with LSS Master capabilities via the CAN Network. The following parameters can be inquired and/or changed by the use of LSS:

- Node-ID of the CANopen Slave
- Bit timing parameters of the physical layer (bit-rate)
- LSS address (Identity Object, Index 1018_h)

By using LSS a LSS slave can be configured for a CANopen network without using any devices such as DIP-switches for setting the parameters.

7.1 LSS Configuration Options

The file ***cbl_conf.h*** holds definitions for the configuration of the LSS module. Please set the symbols to an appropriate value in order to achieve a specific LSS behaviour. For a detailed specification please refer to the HTML documentation.

7.1.1 CBL_LSS_SUPPORT

This symbol is used to enable/disable the LSS capability. If enabled, the file `cbl_lss.c` must be added to the project.

7

8. Flash Programming

Access to the Flash memory is done via three callback functions. They are called when data is written to the CANopen objects $1F50_h$ and $1F51_h$ by a CANopen Program Download Tool.

Function name	Description
`CblMgrBufferStore()`	Store program / data Callback function (cbl_user.c)
`CblMgrFlashDelete()`	Delete program / data in Flash memory Callback function (cbl_user.c)
`CblMgrFlashFinalize()`	Finalize program / data in Flash memory Callback function (cbl_user.c)

Table 8: Callback functions during download

The next figure gives an overview of the involved CANopen objects during program download.

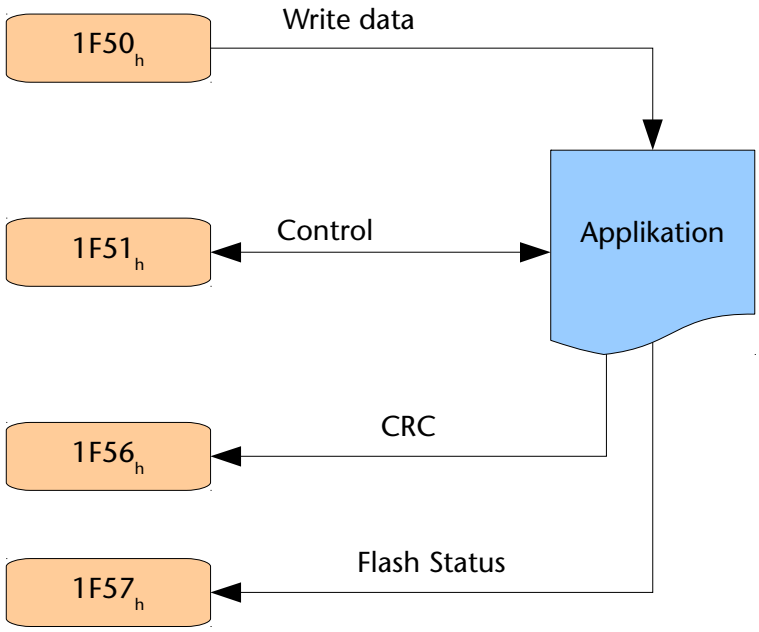

Fig. 3: Involved objects during program download

The following global variables are used inside the callback functions.

Variable	Access	Description
aulIdx1F56_SoftIdentG[]	write	CRC value of program, CANopen access via $1F56_h$
aulIdx1F57_FlashStatusG[]	write	Flash status, CANopen access via $1F56_h$
ubCblMgrBufferStoreG	read / write	write value 0 to mark the end of the flash write operation

Table 9: Usage of global variables inside callback functions

8.1 Erase Flash Memory

The function `CblMgrFlashDelete()` is called by the CANopen Bootloader upon reception of a program erase command in object $1F51_h$. The following pseudo code has to be adopted to the target system.

```
void CblMgrFlashDelete(uint8_t ubProgNumV)
{
   //------------------------------------------------------
   // Todo: check program number & delete flash memory
   //

   //------------------------------------------------------
   // clear software identification, no valid program
   //
   aulIdx1F56_SoftIdentG[ubProgNumV]   = 0x00;
   aulIdx1F57_FlashStatusG[ubProgNumV] =
                    (eCBL_FLASH_FAIL_NO_PROGRAM << 1);
}
```

Example 6: Erase flash memory

The following flowchart shows the interaction upon writing the Erase command or a Manufacturer-specific command to index $1F51_h$ inside the object dictionary.

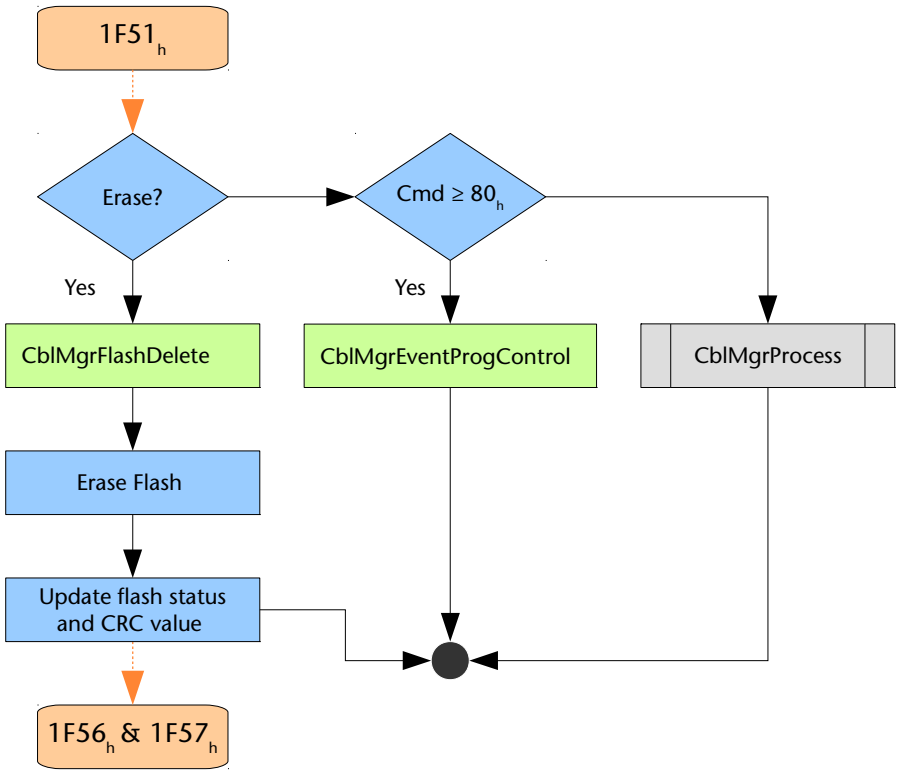

Fig. 4: Callback function for erasing the flash memory

8

8.2 Store Data to Flash Memory

The function **CblMgrBufferStore()** is called by the CANopen Bootloader after a certain amount of data has been copied to the internal RAM of the target system by the CANopen Download Tool. The following pseudo code has to be adopted to the target system.

```
void CblMgrBufferStore(uint8_t ubProgNumV,
                       uint8_t * pubBufferV,
                       uint16_t uwSizeV)
{
   static uint32_t   ulAddrStartS = CBL_APP_START_ADDR;

   //-----------------------------------------------------
   // Todo: check program number
   //

   //-----------------------------------------------------
   // check for data to be stored in flash
   //
   if(ubCblMgrBufferStoreG == 0) return;

   //-----------------------------------------------------
   // test if flash is empty
   //
   if(ulIdx1F57_FlashIdentG ==
         (eCBL_FLASH_FAIL_NO_PROGRAM << 1) )
   {
      ulAddrStartS = CBL_APP_START_ADDR;
      ulIdx1F57_FlashIdentG = eCBL_FLASH_STATUS_PROGRESS;
   }

   //-----------------------------------------------------
   // something went wrong
   //
   if(ulIdx1F57_FlashIdentG != eCBL_FLASH_STATUS_PROGRESS)
   {
      return;
   }


   //-----------------------------------------------------
   // Write data from internal RAM buffer to flash
   //
   if(FlashWrite(ulAddrStartS, pubBufferV, uwSizeV) == FAIL)
   {
     ulIdx1F57_FlashIdentG = eCBL_FLASH_FAIL_WRITE_ERROR << 1;
   }

   ulAddrStartS = ulAddrStartS + uwSizeV;

   //-----------------------------------------------------
   // store operation finished
   //
   ubCblMgrBufferStoreG = 0;
}
```

Example 7: Store data to flash memory

The following flowchart shows the interaction upon writing data to index $1F50_h$ inside the object dictionary.

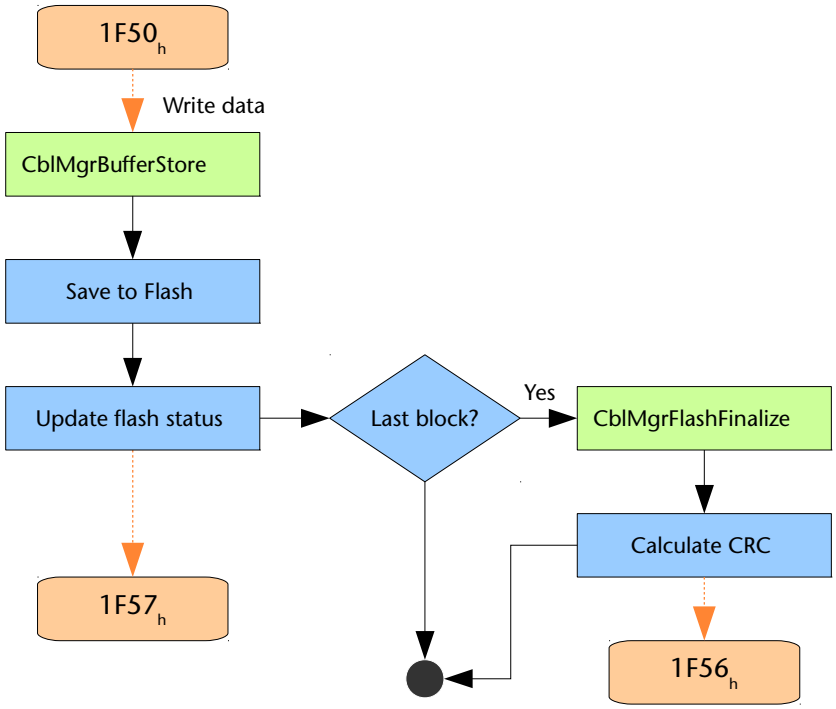

Fig. 5: Callback function for storing data to the flash memory

8

8.3 Finalize Flash Programming

The function `CblMgrFlashFinalize()` is called by the CANopen Bootloader after all data has written to the target system. The following pseudo code has to be adopted to the target system.

```
void CblMgrFlashFinalize(uint8_t ubProgNumV,
                         uint32_t ulSizeV)
{
   uint8_t *   pubMemStartT;     // memory start address
   uint32_t    ulFlashAddrT;     // flash start address


   //-----------------------------------------------------
   // If the flash status is not equal to
   // eCBL_FLASH_STATUS_PROGRESS something went wrong
   //
   if(ulIdx1F57_FlashIdentG != eCBL_FLASH_STATUS_PROGRESS)
   {
      ulIdx1F56_SoftIdentG = 0;
      return;
   }

   //-----------------------------------------------------
   // build checksum over stored data, this can be done
   // also by a manufacturer specific CRC routine
   //
   pubMemStartT = (uint8_t *)CBL_APP_START_ADDR;
   ulIdx1F56_SoftIdentG  = CblMgrBufferCrc(pubMemStartT,
                                           ulSizeV);
   ulIdx1F57_FlashIdentG = eCBL_FLASH_STATUS_OK;


   //-----------------------------------------------------
   // store checksum and length of application data
   //


}
```

Example 8: Finalize the flash programming

8

A Source Code Agreement

This source code agreement (hereinafter - "Agreement”) is made between MicroControl GmbH & Co. KG, Junkersring 23, 53844 Troisdorf, Germany (hereinafter - "MicroControl") and you, the customer (hereinafter - "Licensee"). MicroControl and Licensee agree as follows:

1. DEFINITIONS

1.1 For purposes of this Agreement, the following definitions shall apply:

(a) "Software" shall mean the particular Software product purchased by Licensee from MicroControl.

(b) "Source Code" shall include computer programming code or any computer instructions necessary to compile the Software.

(c) "Derivative Works" means any software programs which are developed by Licensee and which incorporate or contain modifications of any part of Source Code, and including any revision, modification, translation (including compilation or recapitulation by computer), abridgment, condensation, expansion or any other form in which Source Code, may be recast, transformed or adapted.

(d) "Purpose" means the creation of bug-fixes, corrections, enhancements, revisions, modifications and adaptations of Source Code and addition of new user interfaces, features and functionality to the Software.

2. LICENSEE RIGHTS AND RESTRICTIONS

2.1 By completing this Agreement and subject to the restrictions and consideration stated below, MicroControl grants Licensee a non-exclusive, non-transferable, perpetual right to:

(a) use and reproduce as many copies of the Source Code as are reasonably necessary only for the purpose of exercising the rights granted under this Agreement;

(b) modify and create Derivative Works of the Source Code for the Purpose;

(c) use, reproduce, have reproduced, sell (via sub-license), distribute (via sub-license), perform or otherwise transfer (via sub-license), directly or through distributors or resellers, Derivative Works, only in object code format, that are consistent with the Purpose and subject to the reporting and audit provisions of the Agreement.

2.2 No right is granted to Licensee hereunder to permit, authorize, license or sub-license any third party to view or use the Source Code.

2.3 No right is granted to Licensee hereunder to permit, authorize, license or sub-license any company branches or company subsidiary to view or use the Source Code.

2.4 No right is granted to Licensee hereunder to sell, distribute, make available, publish or otherwise transfer the Source Code except as provided in section 2.1 above.

2.5 Licensee shall not use the Source Code for anything other than its intended, legitimate, and legal purpose.

2.6 Licensee shall not employ Source Code in any way that competes either directly or indirectly with MicroControl including but not limited to creation of Derivative Works that compete either directly or indirectly with Software.

2.7 Licensee shall not use the Source Code in any manner not specifically permitted under this Agreement.

2.8 No right is granted under any patents, copyrights, trade secrets, trademarks or other proprietary rights of MicroControl, except as expressly granted herein.

2.9 The terms of this Agreement entitles Licensee to receive support and maintenance services from MicroControl with respect to the Source Code for one year after date of purchase.

3. OWNERSHIP

3.1 MicroControl maintains title and ownership of all copyright interests in the Source Code.

4. CONFIDENTIALITY

4.1 Each party will protect the other's Confidential Information from unauthorized dissemination and use the same degree of care that such party uses to protect its own like information, but in no event less than a reasonable degree of care. Neither party will disclose to third parties the other's Confidential Information without the prior written consent of the other party. Neither party will use the other's Confidential Information for purposes other than those necessary to directly further the purposes of this Agreement. Notwithstanding the foregoing, either party may use or disclose Confidential Information to the extent such party is legally compelled to disclose such Confidential Information provided, however, that prior to any such compelled disclosure, the disclosing party will notify the non-disclosing party and will cooperate fully with the non-disclosing party in protecting against any such disclosure and/or obtaining a protective order narrowing the scope of such disclosure and/or use of the Confidential Information. The parties agree that any breach of this Section would cause irreparable harm to the disclosing party for which monetary damages would not be adequate and therefore, the parties agree that in the event of a breach of this Section 4.1, the disclosing party shall be entitled to equitable relief in addition to any remedies it may have hereunder or at law.

A

4.2 In additional to the provisions of Section 4.1 above, Licensee acknowledges that the Source Code (and to the extent containing MicroControl trade secrets, the Licensee Derivative Works) constitutes a valuable asset of MicroControl and therefore agrees that only the following Licensee employees shall have access to the Source Code and the source code to the Licensee Derivative Works: those employees:

(a) employees who have a need for such access to accomplish the purposes of the distribution rights and license grants specified in Section 2 above; and

(b) with whom Licensee has a legally enforceable obligation that precludes disclosure of third-party proprietary information and is otherwise sufficient to enable Licensee to comply with all the provisions of this Agreement. Licensee shall not grant any other individual or entity access to the Source Code.

4.3 Licensee shall implement reasonable security measures to prevent unauthorized use or disclosure of Source Code. Licensee agrees to segregate all Source Code and Confidential Information from its own confidential information and from the confidential information of others in order to prevent commingling.

4.4 Each party agrees to take appropriate action by instruction, agreement or otherwise with its employees, agents and contractors allowed access to the Confidential Information to satisfy its obligations under this Section 4.

5. LIMITATION ON LIABILITY

5.1 To the maximum extent permitted by applicable law, MicroControl shall not be liable to Licensee for any incidental, consequential, special, punitive or indirect damages, including without limitation, damages for loss of profits, business opportunity, data or use, incurred by Licensee or any third party, even if it has been advised of the possibility of such damages.

6. WARRANTY AND DISCLAIMER

6.1 MicroControl warrants that it has all right, power and authority to enter into this Agreement and to grant the licenses granted hereunder.

6.2 Except as set forth in section 6.1 above, MicroControl makes no representations or warranties with respect to the Source Code. All express or implied representations and warranties, including without limitation any implied warranty of merchantability, of fitness for a particular purpose, of reliability or availability, of accuracy or completeness of responses, of results, of workmanlike effort, of lack of viruses, and of lack of negligence, is hereby expressly disclaimed. Licensee specifically acknowledges that the Source Code is provided "as is" and may have bugs, errors, defects or deficiencies.

7. INDEMNITY

7.1 Licensee agrees to defend, indemnify and hold harmless MicroControl from and against any damages, costs and expenses (including, without limitation, reasonable attorneys fees and costs) arising from or relating to any third party claims, actions or demands that the sale, distribution or other transfer of any Derivative Works by Licensee or its distributors or resellers infringes the intellectual property rights of any third party.

8. TERMINATION.

8.1 This Agreement is in effect so long as Licensee holds any copy of the Source Code on any Licensee computer or storage media either onsite or offsite.

8.2 Upon termination or expiration of this Agreement for any reason whatsoever, Licensee shall immediately:

(a) cease all use of Product Source Code;

(b) make all reasonable effort to destroy and/or remove all copies of Source Code from Licensee computers and storage media; and

(c) return all Software, Source Code, Documentation, Confidential Information, and the source code to all Licensee Derivative Works and all related materials and copies thereof to MicroControl. In addition to the foregoing, Licensee agrees that it shall not, following termination of this Agreement, act in any way to damage the reputation or goodwill of MicroControl or any Software, Licensee Derivative Work or other product.

8.3 Except as otherwise expressly provided herein, upon the expiration or termination of this Agreement Licensee shall not be entitled to, and to the fullest extent permitted by law waives, any statutorily prescribed or other compensation, reimbursement or damages for loss of goodwill, clientele, prospective profits, investments or anticipated sales or commitments of any kind.

9. MISCELLANEOUS

9.1 Assignment and Effect: This Agreement shall inure to the benefit of and be binding upon both parties, as well as their employees, employers, agents, parents, subsidiaries, representatives, licensees, and assigns.

9.2 Modifications: There will be no modifications, alterations, or amendments to this Agreement, unless both parties agree in writing.

9.3 Governing Law: This Agreement shall be governed by and construed under the laws of the Federal Republic of Germany.

9.4 Jurisdiction and Venue: Should any dispute arise under the terms of this Agreement, such dispute will finally be solved under the procedure established by the laws of the Federal Republic of Germany in the German court according to the place of domicile of MicroControl.

9.5 Transfer of Rights: Without prejudice to any other rights, MicroControl shall have the right to transfer any rights and/or obligations hereunder to any third party.

A

B. Index

D

Device Profile **32**

E

EMCY **37**
EMCY Producer Functions **38**
    CblEmcySend **38**

L

LSS **39**

M

Manager **15**
Manager Functions **18**
    CblMgrBufferStore **19**
    CblMgrFlashDelete **21**
    CblMgrFlashFinalize **21**
    CblMgrInit **22**
    CblMgrOnBusOff **20**
    CblMgrRelease **23**
    CblMgrStart **23**
    ComTmrEvent **24**

N

NMT **25**
NMT Functions **25**

O

Object
    1000h **32**
    1001h **32**
    1008h **33**
    1009h **33**
    100Ah **33**
    1018h **34**

S

SDO **29**

B

B

Disclaimers

Life support — Products and software described in this manual are not designed for use in life support appliances, devices, or systems where malfunction of these products can reasonably be expected to result in personal injury. MicroControl customers using or selling these products for use in such applications do so at their own risk and agree to fully indemnify MicroControl for any damages resulting from such application.

Right to make changes — MicroControl reserves the right to make changes in the products - including circuits and/or software - described or contained herein in order to improve design and/or performance. MicroControl assumes no responsibility or liability for the use of any of these products, conveys no licence or title under any patent, copyright, or mask work right to these products, and makes no representations or warranties that these products are free from patent, copyright, or mask work right infringement, unless otherwise specified.

Copyright

No part of this functional specification may be copied, transmitted or stored in a retrieval system or reproduced in any way including, but not limited to, photography, magnetic, optic or other recording means, without prior written permission from MicroControl GmbH & Co. KG.

© 2014 MicroControl GmbH & Co. KG, Troisdorf

MicroControl
Systemhaus für Automatisierung

MicroControl GmbH & Co. KG
Junkersring 23
D-53844 Troisdorf
Fon: +49 / 2241 / 25 65 9 - 0
Fax: +49 / 2241 / 25 65 9 - 11
http://www.microcontrol.net