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 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
Lindlaustraße 2c
D-53842 Troisdorf
Fon: +49 / 2241 / 25 65 9 - 0
Fax: +49 / 2241 / 25 65 9 - 11
http://www.microcontrol.net
## Contents

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

2. **CANopen Master Overview** .............................. 9  
   2.1 Naming Conventions ......................... 10  
   2.2 Message Distribution .................... 11  
   2.3 File Structure ............................... 12  
   2.4 Configuration Options ............... 13  
   2.5 Initialisation Process ................. 14  

3. **CANopen Master 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 Configuration Options ............. 25  
   4.2 NMT Functions ............................... 26  

5. **Service Data Objects - SDO** .......................... 31  
   5.1 SDO Client Configuration Options ...... 31  
   5.2 SDO Client Functions .................... 32  
   5.3 SDO Server Configuration Options ...... 39  
   5.4 SDO Server Functions .................... 40  

6. **Process Data Objects - PDO** .......................... 41  
   6.1 PDO Configuration Options .............. 41  
   6.2 PDO Functions ............................... 42  

7. **Emergency Service - EMCY** .......................... 49  
   7.1 EMCY Producer Configuration Options .. 49  
   7.2 EMCY Producer Functions ............... 50  
   7.3 EMCY Consumer Configuration Options .. 53  
   7.4 EMCY Consumer Functions ................ 54
8. **Synchronisation Service - SYNC** .................................. 59
   8.1  SYNC Configuration Options  .............................. 59
   8.2  SYNC Functions  ............................................. 60

9. **Layer Setting Services - LSS** .................................. 63
   9.1  LSS Configuration Options  .............................. 63
   9.2  LSS Functions  .............................................. 64

A  **Source Code Agreement** .......................................... 71

B. **Index** ................................................................. 75
1. Scope

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

The CANopen Master Protocol Stack is independent from the used CAN hardware and operating system. CAN hardware is accessed 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 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

1.2 Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN</td>
<td>Controller area network</td>
</tr>
<tr>
<td>CAN-ID</td>
<td>CAN identifier</td>
</tr>
<tr>
<td>COB</td>
<td>Communication object</td>
</tr>
<tr>
<td>COB-ID</td>
<td>Communication object identifier</td>
</tr>
<tr>
<td>CRC</td>
<td>Cyclic redundancy check</td>
</tr>
<tr>
<td>LSB</td>
<td>Least significant bit/byte</td>
</tr>
<tr>
<td>MSB</td>
<td>Most significant bit/byte</td>
</tr>
<tr>
<td>OSI</td>
<td>Open systems interconnection</td>
</tr>
<tr>
<td>RTR</td>
<td>Remote transmission request</td>
</tr>
</tbody>
</table>
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

The CAN (Controller Area Network) protocol 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 stipulated during the system design phase 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 is 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.
1.5 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 Master Overview

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

Fig. 1: CANopen Master overview

The CANopen Master Protocol Stack uses a well-defined 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/.

The CANopen Master Protocol Stack supports up to 250 independent CAN networks. The functionality of the different CANopen services can be configured individually in order to achieve an optimal performance for different platforms and applications.
2.1 Naming Conventions

All functions, structures, definitions and constant values have the prefix "Com". The following table shows the used nomenclature:

<table>
<thead>
<tr>
<th>CANopen Master stack</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functions</td>
<td>Com&lt;service&gt;&lt;name&gt;</td>
</tr>
<tr>
<td>Enumeration</td>
<td>eCOM_&lt;name&gt;</td>
</tr>
<tr>
<td>Structures</td>
<td>Com&lt;name&gt;_s</td>
</tr>
<tr>
<td>Defines</td>
<td>COM_&lt;service&gt;_&lt;name&gt;</td>
</tr>
<tr>
<td>Error Codes</td>
<td>eCOM_ERR_&lt;name&gt;</td>
</tr>
</tbody>
</table>

*Table 1: Naming conventions*
2.2 Message Distribution

The message distribution is responsible for reading and writing CAN messages between the CANopen Master protocol stack and the CAN bus. The CANpie API [4] and its buffer concept are 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 message and event handlers with callback functions
### 2.3 File Structure

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

<table>
<thead>
<tr>
<th>File</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>com_cli_cln.c / h</td>
<td>Command line interface (CLI) client</td>
</tr>
<tr>
<td>com_cli_srv.c / h</td>
<td>Command line interface (CLI) server</td>
</tr>
<tr>
<td>com_conf.h</td>
<td>CANopen Master configuration file</td>
</tr>
<tr>
<td>com_dict.c / h</td>
<td>Object dictionary</td>
</tr>
<tr>
<td>com_emcy.c / h</td>
<td>Emergency service</td>
</tr>
<tr>
<td>com_led.c / h</td>
<td>LED support</td>
</tr>
<tr>
<td>com_lss.c / h</td>
<td>Layer Settings Service (LSS)</td>
</tr>
<tr>
<td>com_mgr.c / h</td>
<td>CANopen Master manager</td>
</tr>
<tr>
<td>com_mobj.c / h</td>
<td>Manufacturer objects</td>
</tr>
<tr>
<td>com_nmt.c / h</td>
<td>Network management (NMT)</td>
</tr>
<tr>
<td>com pdo.c / h</td>
<td>Process data objects (PDO) service</td>
</tr>
<tr>
<td>com_sdo_cln.c / h</td>
<td>Service data objects (SDO) client</td>
</tr>
<tr>
<td>com_sdo_srv.c / h</td>
<td>Service data objects (SDO) server</td>
</tr>
<tr>
<td>com_sync.c / h</td>
<td>SYNC service</td>
</tr>
<tr>
<td>com_tmr.c / h</td>
<td>Timer services</td>
</tr>
<tr>
<td>com_user.c</td>
<td>Application funtions / event handler</td>
</tr>
</tbody>
</table>

*Table 2: CANopen Master stack files*
2.4 Configuration Options

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

2.4.1 COM_NET_MAX

This symbol defines the maximum number of CANopen Master networks in a system. The value is typically equal to the number of physical CAN interfaces.

2.4.2 COM_NODE_MAX

This symbol defines the maximum number of CANopen devices in a network. The default value is 127, which is also the maximum value. A lower value affects the required memory resources (lower RAM requirements).

2.4.3 COM_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.

2.4.4 COM_SDO_CLIENT_MAX

The symbol defines the number of SDO clients that are supported. Possible values are 1 to 127. The number affects the required CAN message buffers and the code size.

2.4.5 COM_PDO_RCV_MAX

This symbol defines the total number of receive PDOs available by the CANopen Master stack. The number may vary between 0 and 512. The number affects the required RAM size.

2.4.6 COM_PDO_TRM_MAX

This symbol defines the total number of transmit PDOs available by the CANopen Master stack. The number may vary between 0 and 512. The number affects the required RAM size.
2.5 Initialisation Process

The CANopen Master is initialised with `ComMgrInit()`. This routine will setup the CAN controller and initialise all necessary services. Support of CANopen services and their behaviour can be configured using the `com_conf.h` file. Afterwards the stack can be started by calling the `ComMgrStart()` to start running the CANopen Master.

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

```c
void MyComInit(void)
{
    // Initialise the CANopen Master stack with address 1
    // Bitrate = 500kBit/s
    //
    ComMgrInit(CP_CHANNEL_1, eCOM_NET_1,
                CP_BAUD_500K,
                1,
                eCOM_MODE_NMT_MASTER);

    // start the CANopen master stack
    //
    ComMgrStart(eCOM_NET_1);

    // now the CANopen Master Stack is initialised and
    // has to be triggered by calling ComMgrTimerEvent()
}
```

*Example 1: Initialisation process of the CANopen Master Protocol Stack*

The initialisation functions of CANopen Master Protocol Stack have to be executed before any other services.
3. CANopen Master Manager

The CANopen Master Manager covers the initialisation and control of the protocol stack. The module also manages the initialisation of CAN interfaces via the CANpie driver.

3.1 Initialisation

The CANopen Master stack is initialised by calling the two functions ComMgrInit() and ComMgrStart().

```
// Initialise the CANopen Master stack with address 1
// Bitrate = 500K
//
ComMgrInit(CP_CHANNEL_1, eCOM_NET_1, CP_BAUD_500K, 1,
eCOM_MODE_NMT_MASTER);

// start the CANopen master stack
//
ComMgrStart(eCOM_NET_1);
```

Example 2: Initialization of CANopen Master stack

After calling ComMgrStart() the CANopen Master stack is running and a Bootup message is send on the CAN bus.

In order to have periodical functions available (e.g. heartbeat), it is necessary to call the function ComMgrTimerEvent() cyclically. The cycle time is defined in multiples of one microsecond by COM_TIMER_PERIOD inside com_conf.h and must match the time cycle period.

Typically ComMgrTimerEvent() is called from a timer interrupt, but it is also possible to call it from main loop. This behaviour is controlled by COM_TMR_INT defined in com_conf.h.

The example below 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).
Example 3: Complete generic initialisation of CANopen Master stack

```c
// Initialise the target CPU
//
McCpuInit();

// Initialise the CANopen Master stack with address 1
// Bitrate = 500K
//
ComMgrInit(CP_CHANNEL_1, eCOM_NET_1, CP_BAUD_500K, 1,
          eCOM_MODE_NMT_MASTER);

// Initialise the timer resource on the target CPU
//
McTmrInit();

// Start the timer event function
//
McTmrFunctionInit(ComMgrTimerEvent, 1, eTMR_CTRL_START);

// start the CANopen master stack
//
ComMgrStart(eCOM_NET_1);

// this is the main loop of the embedded application
//
while (1)
{
    // if the target system can not call the
    // ComMgrTimerEvent() function inside the timer
    // interrupt, this is performed here inside
    // the main loop
    //
    #if COM_TMR_INT == 0
    if(ulLastTickValueT != McTmrTick())
    {
        ComMgrTimerEvent();
        ulLastTickValueT = McTmrTick();
    }
    #endif
}  // end while (1)
```
3.2 Manager Configuration Options

The file `com_conf.h` holds 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 COM_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 COM_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 `ComMgrTimerEvent()` is called within the timer interrupt.
3.3 Manager Functions

The manager functions (prefix ComMgr) provide general control over the CANopen Master protocol stack.

<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ComMgrInit()</td>
<td>Initialise protocol stack and CAN interface</td>
</tr>
<tr>
<td>ComMgrMasterInit()</td>
<td>Modify protocol stack behaviour by application, Callback function (com_user.c)</td>
</tr>
<tr>
<td>ComMgrEventBus()</td>
<td>CAN bus status change, Callback function (com_user.c)</td>
</tr>
<tr>
<td>ComMgrRelease()</td>
<td>Shutdown protocol stack and CAN interface</td>
</tr>
<tr>
<td>ComMgrStart()</td>
<td>Start CANopen Master</td>
</tr>
<tr>
<td>ComMgrStop()</td>
<td>Start CANopen Master</td>
</tr>
<tr>
<td>ComMgrTimerEvent()</td>
<td>Handler for periodic services</td>
</tr>
</tbody>
</table>

*Table 3: Functions of COM Manager*
### 3.3.1 ComMgrInit

**Syntax**

```c
ComStatus_tv ComMgrInit(
    uint8_t ubCanIfV,
    uint8_t ubNetV,
    uint8_t ubBaudSelV,
    uint8_t ubNodeIdV,
    uint32_t ulModeV)
```

**Function**

Initialise CANopen Master stack

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

The function assigns the logical CANopen network number `ubNetV` to the CAN interface given by `ubCanIfV`. The value for `ubNetV` can be taken from the `ComNet_e` enumeration. The maximum value is given by the symbol `COM_NET_MAX`.

The parameter `ubBaudSelV` defines the bitrate on the selected CAN interface. Valid values for `ubBaudSelV` are defined by the `CP_BAUD` enumeration. The node-ID of the CANopen Master protocol stack is given by the parameter `ubNodeIdV`, it must be in the range between 1 and `COM_NODE_MAX`.

The initial mode parameter `ulModeV` defines the startup behaviour. Valid values are defined in the enumeration `ComMode_e`.

For the usage of this function, please refer to the example in "Initialisation" on page 15.

**Parameters**

- **ubCanIfV**: CAN interface index
- **ubNetV**: CANopen Network channel
- **ubBaudSelV**: Bit-rate value
- **ubNodeIdV**: Node-ID value
- **ulModeV**: Initial mode

**Return value**

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.
### 3.3.2 ComMgrMasterInit

**Syntax**

```c
ComStatus_tv ComMgrMasterInit(
    uint8_t        ubNetV)
```

**Function**

Initialise CANopen Master settings

This function is called by `ComMgrInit()` during the initialisation phase. It is responsible for setting internal values of the Master.

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

**Parameters**

- `ubNetV` CANopen Network channel

**Return value**

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.

### 3.3.3 ComMgrEventBus

**Syntax**

```c
void ComMgrEventBus(
    uint8_t        ubNetV,
    CpState_ts *   ptsBusStateV)
```

**Function**

COM Manager event callback

This function is called by the stack on a bus status change, e.g. the CAN interface changes from error active to error passive state.

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

**Parameters**

- `ubNetV` CANopen Network channel
- `ptsBusStateV` Pointer to `CpState_ts` structure

**Return value**

None
### 3.3.4 ComMgrRelease

**Syntax**

```c
ComStatus_tv ComMgrRelease(
    uint8_t ubNetV)
```

**Function**

Shutdown the CANopen Master

This function performs a shutdown of the CANopen Master.

**Parameters**

- `ubNetV`: CANopen Network channel

**Return value**

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.

### 3.3.5 ComMgrStart

**Syntax**

```c
ComStatus_tv ComMgrStart(
    uint8_t ubNetV)
```

**Function**

Start the CANopen Master

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

![Tip]

The internal NMT state of the CANopen Master protocol stack (Pre-Operational, Operational) depends on the parameter `ulModeV` of the function `ComMgrInit()`.

**Parameters**

- `ubNetV`: CANopen Network channel

**Return value**

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.
3.3.6 ComMgrStop

Syntax

```c
ComStatus_tv ComMgrStop(
    uint8_t    ubNetV)
```

Function

Stop the CANopen Master

This function will stop any CAN communication of the CANopen Master protocol stack.

Parameters

- `ubNetV` CANopen Network channel

Return value

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.
3.3.7 ComMgrTimerEvent

Syntax

```c
void ComMgrTimerEvent(
    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).

The cycle time of the timer resource is defined via the symbol COM_TIMER_PERIOD.

```c
/
// Timer interrupt service routine
//
// Call CANopen stack timer function
void MyTimerInterrupt(void)
{
    //... timer services of application ...
    ComMgrTimerEvent();
    //... retrigger the timer
}
/
```

Example 4: Example routine for ComMgrTimerEvent()

Parameters

None

Return value

None
4. Network Management - NMT

The network management (NMT) is CANopen device oriented and follows a master-slave structure. NMT objects are used for executing 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/.

This module holds functions for the protocol stack to behave in both ways: as Master or as Slave. If the protocol stack is started in Slave mode (refer to “ComMgrInit” on page 19), the protocol stack must react on NMT messages. If started in Master mode, the protocol stack holds the NMT resource and is allowed to send NMT messages.

4.1 NMT Configuration Options

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

4.1.1 COM_NMT_HEARTBEAT_CONSUMER

With this symbol it is possible to enable the NMT Heartbeat consumer service. It is enabled by default. Setting this symbol to 0 means that index 1016h is not supported and the NMT state of other nodes in the network cannot be tested.

4.1.2 COM_NMT_FLYING_MASTER

With this symbol it is possible to enable the NMT Flying Master functionality and protocol.

4.1.3 COM_NMT_MASTER_PRIORITY

This symbol defines the NMT master priority for the flying master negotiation procedure.
4.2 NMT Functions

The Network Management functions of the CANopen Master protocol stack have the prefix ComNmt and are located in the file com_nmt.c within the source directory.

<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ComNmtEventActiveMaster()</td>
<td>Flying master event, Callback function (com_user.c)</td>
</tr>
<tr>
<td>ComNmtEventHeartbeat()</td>
<td>NMT heartbeat event, Callback function (com_user.c)</td>
</tr>
<tr>
<td>ComNmtEventIdCollision()</td>
<td>CAN identifier collision, Callback function (com_user.c)</td>
</tr>
<tr>
<td>ComNmtEventStateChange()</td>
<td>NMT state change event, Callback function (com_user.c)</td>
</tr>
<tr>
<td>ComNmtGetNodeState()</td>
<td>Retrieve NMT state of a CANopen device</td>
</tr>
<tr>
<td>ComNmtSetHbConsTime()</td>
<td>Set heartbeat consumer time of a CANopen device</td>
</tr>
<tr>
<td>ComNmtSetHbProdTime()</td>
<td>Set heartbeat producer time of a CANopen device</td>
</tr>
<tr>
<td>ComNmtSetInhibit()</td>
<td>Set inhibit time for NMT messages</td>
</tr>
<tr>
<td>ComNmtSetNodeState()</td>
<td>Change NMT state of a CANopen device</td>
</tr>
</tbody>
</table>

Table 4: NMT functions
4.2.1 ComNmtEventHeartbeat

Syntax
void ComNmtEventHeartbeat(
    uint8_t ubNetV,
    uint8_t ubNodeIdV)

Function
This function is called by the stack on a heartbeat consumer timeout. The heartbeat consumer time for a CANopen device is configured by the function ComNmtSetHbConsTime(). Since the behaviour of this function is application specific, the implementation is available in the file com_user.c.

Parameters
ubNetV CANopen Network channel
ubNodeIdV Node-ID value

Return value
None

4.2.2 ComNmtEventIdCollision

Syntax
void ComNmtEventIdCollision(
    uint8_t ubNetV)

Function
This function is called by the stack on a CAN identifier collision. Since the behaviour of this function is application specific, the implementation is available in the file com_user.c.

Parameters
ubNetV CANopen Network channel

Return value
None
4.2.3 **ComNmtEventStateChange**

**Syntax**

```c
void ComNmtEventStateChange(
    uint8_t ubNetV,
    uint8_t ubNodeIdV,
    uint8_t ubNmtStateV)
```

**Function**

This function is called by the stack on a CANopen device NMT state change event. Possible values for `ubNmtStateV` are listed in the `ComNmtState_e` enumeration.

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

**Parameters**

- `ubNetV` CANopen Network channel
- `ubNodeIdV` Node-ID value
- `ubNmtStateV` New NMT State

**Return value**

None

4.2.4 **ComNmtGetNodeState**

**Syntax**

```c
ComStatus_tv ComNmtGetNodeState(
    uint8_t ubNetV,
    uint8_t ubNodeIdV,
    uint8_t *pubStateV)
```

**Function**

Retrieve NMT state of CANopen device

This function retrieves the NMT state of the CANopen device with the Node-ID `ubNodeIdV`. Possible values for `ubNodeIdV` are 1 to `COM_NODE_MAX`. The value is passed to the pointer `pubStateV` and is taken from the `ComNmtState_e` enumeration.

**Parameters**

- `ubNetV` CANopen Network channel
- `ubNodeIdV` Node-ID value
- `pubStateV` Pointer to NMT state variable

**Return value**

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.
4.2.5 ComNmtSetHbConsTime

Syntax

```c
ComStatus_tv ComNmtSetHbConsTime(
    uint8_t ubNetV,
    uint8_t ubNodeIdV,
    uint16_t uwTimeV)
```

Function

This function sets the heartbeat consumer time for the CANopen device with the Node-ID `ubNodeIdV`. Possible values for `ubNodeIdV` are 1 to `COM_NODE_MAX`. The value passed to `uwTimeV` is given in multiples of 1 millisecond.

If the parameter `ubNodeIdV` is equal to the Node-ID of the CANopen Master, the function will return an error (`eCOM_ERR_NODE_ID`).

Parameters

- `ubNetV` CANopen Network channel
- `ubNodeIdV` Node-ID value
- `uwTimeV` Heartbeat timer value

Return value

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.

4.2.6 ComNmtSetHbProdTime

Syntax

```c
ComStatus_tv ComNmtSetHbProdTime(
    uint8_t ubNetV,
    uint16_t uwTimeV)
```

Function

This function sets the heartbeat producer time for the CANopen Master. The value passed to `uwTimeV` is given in multiples of 1 millisecond.

Parameters

- `ubNetV` CANopen Network channel
- `uwTimeV` Heartbeat timer value

Return value

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.
### 4.2.7 ComNmtSetNodeState

**Syntax**

```c
ComStatus_tv ComNmtSetNodeState(  
    uint8_t ubNetV,  
    uint8_t ubNodeIdV,  
    uint8_t ubStateV)
```

**Function**

This function changes the NMT state of the CANopen device with the Node-ID `ubNodeIdV`. Possible values for `ubNodeIdV` are 0 to `COM_NODE_MAX`. A value of 0 addresses all devices in the network.

Passing the address of the CANopen Master to `ubNodeIdV` will change the NMT state of the Master. The value passed to `ubStateV` is taken from the `ComNmtState_e` enumeration.

**Parameters**

- `ubNetV` CANopen Network channel
- `ubNodeIdV` Node-ID value
- `ubStateV` New NMT state

**Return value**

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.
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 which data set is to be transferred via a multiplexor (index and sub-index of the Object Dictionary). The contents of the data set are defined within the Object Dictionary /1/.

The communication timeout between SDO client request and SDO server response is device/application specific. The initial SDO communication timeout for the client is set to 50 milliseconds. The connection timeout value can be changed by the ComSdoSetTimeout() function.

5.1 SDO Client Configuration Options

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

5.1.1 COM_SDO_CLIENT_MAX

The symbol defines the number of SDO clients supported. Possible values are 1 to 127. The number affects the required CAN message buffers and the code size.
5.2 SDO Client Functions

The SDO client functions of the CANopen Master protocol stack have the prefix `ComSdo` and are located in the file `com_sdo_cln.c` within the source directory.

<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ComSdoGetAbortCode()</code></td>
<td>Get SDO abort code</td>
</tr>
<tr>
<td><code>ComSdoGetStatus()</code></td>
<td>Get status of a SDO transmission</td>
</tr>
<tr>
<td><code>ComSdoReadData()</code></td>
<td>Read SDO data from CANopen device</td>
</tr>
<tr>
<td><code>ComSdoReadObject()</code></td>
<td>Read multiple objects from CANopen device</td>
</tr>
<tr>
<td><code>ComSdoSetConfiguration()</code></td>
<td>Configure SDO client</td>
</tr>
<tr>
<td><code>ComSdoSetTimeout()</code></td>
<td>Set timeout value for SDO communication</td>
</tr>
<tr>
<td><code>ComSdoWriteData()</code></td>
<td>Write SDO data to CANopen device</td>
</tr>
<tr>
<td><code>ComSdoWriteObject()</code></td>
<td>Write multiples objects to CANopen device</td>
</tr>
</tbody>
</table>

*Table 5: SDO Client functions*
5.2.1 ComSdoGetAbortCode

Syntax

```c
ComStatus_tv ComSdoGetAbortCode(
    uint8_t ubNetV,
    uint8_t ubSdoV,
    uint32_t * pulAbortV)
```

Function

This function returns the last SDO abort code, if any occurred. The SDO client connection is selected by the index `ubSdoV`. A negative return value indicates an error. The error values are taken from the `ComErr_e` enumeration. The return value `eCOM_ERR_OK` indicates that there was a SDO abort on the selected SDO client, which is copied to the variable pointed by `pulAbortV`.

Parameters

- `ubNetV` CANopen Network channel
- `ubNodeIdV` Node-ID value
- `pulAbortV` Pointer to variable for abort code

Return value

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned. A positive return value indicates the SDO client is in use. The positive return value taken from the `ComSdoStat_e` enumeration.

5.2.2 ComSdoGetStatus

Syntax

```c
ComStatus_tv ComSdoGetStatus(
    uint8_t ubNetV,
    uint8_t ubSdoV)
```

Function

This function returns the status of a SDO transmission for the SDO client connection given by its index `ubSdoV`.

Parameters

- `ubNetV` CANopen Network channel
- `ubSdoV` SDO client index

Return value

On failure a negative value from the `ComErr_e` enumeration is returned. Positive return values are taken from the `ComSdoStat_e` enumeration and indicate the status of a SDO client. The return value `eCOM_SDO_STAT_EMPTY` indicates that the SDO client is available for a read/write operation.
5.2.3 ComSdoReadData

Syntax

```c
ComStatus_tv ComSdoReadData(
    uint8_t ubNetV,
    uint8_t ubSdoV,
    uint8_t ubNodeIdV,
    uint16_t uwIndexV,
    uint8_t ubSubIndexV,
    void * pvdDataV,
    uint32_t * pulDataSizeV,
    uint32_t * pulErrCodeV)
```

Function

This function will start the read request for an object that is defined by the parameters `uwIndexV` and `ubSubIndexV` for the Node-ID `ubNodeIdV`. The SDO client can be selected via the parameter `ubSdoV` in the range from 0 to `COM_SDO_CLIENT_MAX - 1`.

The function is non-blocking, i.e. it returns immediately after the CAN message transmission is requested. In order to check if data has been transferred by the SDO server the `ComSdoGetStatus()` function should be used. The pointer to the data size variable `pulDataSizeV` is used to indicate the size of the read buffer. If the buffer size is too small for the read operation the SDO connection will be terminated and the SDO status is set to `eCOM_SDO_STAT_SIZE`. The number of bytes read from the SDO server is copied in the data size variable on operation success.

The parameter `pulErrCodeV` points to a variable for the possible SDO abort code. If the application does not require an abort code, the pointer can be set to `0L`.

Parameters

- `ubNetV`: CANopen Network channel
- `ubSdoV`: SDO client index
- `ubNodeIdV`: Node-ID value
- `uwIndexV`: Index of object
- `ubSubIndexV`: Sub-Index of object
- `pvdDataV`: Pointer to data
- `pulDataSizeV`: Pointer to data size variable (bytes)
- `pulErrCodeV`: Pointer to variable for SDO abort code

Return value

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.
The following example shows a read operation from a CANopen device with node-ID 3. Data is read from index 1000h, sub-index 0.

```c
uint32_t ulIndex1000T;
uint32_t ulDataSizeT;
uint32_t ulErrCodeT;

// test if SDO client 0 is available
if (ComSdoGetStatus(eCOM_NET_1, 0) == eCOM_SDO_STAT_EMPTY) {
    ComSdoReadData(eCOM_NET_1, 0, 3, 0x1000, 0x00,
                    &ulIndex1000T, &ulDataSizeT, &ulErrCodeT);
}
```

**Example 5:** Read SDO Data from a CANopen device

### 5.2.4 ComSdoReadObject

**Syntax**

```c
ComStatus_tv ComSdoReadObject(
    uint8_t ubNetV,
    uint8_t ubSdoV,
    uint8_t ubNodeIDV,
    CoObject_ts * ptsCoObjV,
    uint32_t * pulObjCntV)
```

**Function**

This function will start the read request for an object that is defined by the structure `CoObject_ts`. User can also request an array of objects to be read.

The function is not blocking and has the same behaviour as `ComSdoReadData()`.

**Parameters**

- `ubNetV`: CANopen Network channel
- `ubSdoV`: SDO client index
- `ubNodeIDV`: Node-ID value
- `ptsCoObjV`: Pointer to CANopen object entry
- `pulObjCntV`: Pointer to variable for object count

**Return value**

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.
5.2.5 ComSdoSetConfiguration

Syntax

ComStatus_tv ComSdoSetConfiguration(
    uint8_t ubNetV,
    uint8_t ubSdoV,
    uint8_t ubConfigV)

Function

This function is used to change the behaviour of the SDO client. The possible configuration values are defined via the ComSdoConfig_e enumeration.

Parameters

ubNetV CANopen Network channel
ubSdoV SDO client index
ubConfigV SDO client configuration value

Return value

On success the value eCOM_ERR_OK is returned. On failure a negative value from the ComErr_e enumeration is returned.

5.2.6 ComSdoSetTimeout

Syntax

ComStatus_tv ComSdoSetTimeout(
    uint8_t ubNetV,
    uint8_t ubSdoV,
    uint16_t uwTimeV)

Function

This function sets a timeout value for a SDO communication. If the response to a SDO request by the master takes longer than this timeout value, the state of the communication will be changed to eCOM_SDO_STAT_TIMEOUT. The state of the SDO communication can be evaluated by calling ComSdoGetStatus(). Please note that the timeout value is passed in multiples of 1 millisecond. The timeout value supplied to this function will be rounded towards the next lower value if necessary (depends on timer granularity).

Parameters

ubNetV CANopen Network channel
ubSdoV SDO client index
uwTimeV timeout value in milliseconds

Return value

On success the value eCOM_ERR_OK is returned. On failure a negative value from the ComErr_e enumeration is returned.
5.2.7 ComSdoWriteData

Syntax

```c
ComStatus_tv ComSdoWriteData(
    uint8_t ubNetV,
    uint8_t ubSdoV,
    uint8_t ubNodeIdV,
    uint16_t uwIndexV,
    uint8_t ubSubIndexV,
    void * pvdDataV,
    uint32_t * pulDataSizeV,
    uint32_t * pulErrCodeV)
```

Function

This function will start the write request for an object that is defined by the parameters `uwIndexV` and `ubSubIndexV` for the Node-ID `ubNodeIdV`. The SDO client can be selected via the parameter `ubSdoV` in the range from 0 to `COM_SDO_CLIENT_MAX - 1`.

The function is non-blocking, i.e. it returns immediately after the CAN message transmission is requested. In order to check if data has been transferred by the SDO server the `ComSdoGetStatus()` function should be used. The pointer to the data size variable `pulDataSizeV` is used to indicate the size of the data to be written. If the data size does not match, the write operation will be terminated and the SDO status is set to `eCOM_SDO_STAT_SIZE`.

The parameter `pulErrCodeV` points to a variable for the possible SDO abort code. If the application does not required an abort code, the pointer can be set to `0L`.

Parameters

- `ubNetV`: CANopen Network channel
- `ubSdoV`: SDO client index
- `ubNodeIdV`: Node-ID value
- `uwIndexV`: Index of object
- `ubSubIndexV`: Sub-Index of object
- `pvdDataV`: Pointer to data
- `pulDataSizeV`: Pointer to data size variable (bytes)
- `pulErrCodeV`: Pointer to variable for SDO abort code

Return value

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.

The following example shows a write operation to a CANopen device with node-ID 3. Data is written to index 1017h, sub-index 0.
5.2.8 ComSdoWriteObject

Syntax

```c
ComStatus_tv ComSdoWriteObject(
    uint8_t ubNetV,
    uint8_t ubSdoV,
    uint8_t ubNodeIdV,
    CoObject_ts * ptsCoObjV,
    uint32_t * pulObjCntV)
```

Function

This function will start the write request for an object that is defined by the structure `CoObject_ts`. User can also request an array of objects to be written. The function is not blocking and has the same behaviour as `ComSdoWriteData()`.

Parameters

- `ubNetV` CANopen Network channel
- `ubSdoV` SDO client index
- `ubNodeIdV` Node-ID value
- `ptsCoObjV` Pointer to CANopen object entry
- `pulObjCntV` Pointer to variable for object count

Return value

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.

```c
uint16_t ulIndex1017T = 500;
uint32_t ulDataSizeT = sizeof(ulIndex1017T);
uint32_t ulErrCodeT;

// test if SDO client 0 is available
if( ComSdoGetStatus(eCOM_NET_1, 0) == eCOM_SDO_STAT_EMPTY) {
    ComSdoWriteData( eCOM_NET_1, 0, 3, 0x1017, 0x00, &ulIndex1017T, &ulDataSizeT, &ulErrCodeT);
}
```

Example 6: Write SDO Data to a CANopen device
5.3 SDO Server Configuration Options

The file `com_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.

5.3.1 COM_SDO_SRV

The symbol defines if a SDO server is supported. If the SDO server is disabled, an object dictionary for the CANopen Master stack is not generated.

5.3.2 COM_SDO_SRV_SEGMENTED

The symbol defines if the segmented SDO transfer is supported. If segmented SDOs are not supported, the code size can be reduced. However, segmented SDOs are required if the data type STRING has to be supported with string length greater than 4.

5.3.3 COM_SDO_SRV_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.
5.4 SDO Server Functions

The SDO server functions of the CANopen Master protocol stack have the prefix ComSdoSrv and are located in the file `com_sdo_srv.c` within the source directory.
6. Process Data Objects - PDO

The real-time data transfer is performed by means of "Process Data Objects (PDO)". The transfer of PDOs is performed with no protocol overhead.

The PDOs correspond to entries in the device Object Dictionary and provide the interface to the application objects. Data type and mapping of application objects into a PDO are determined by a corresponding default PDO mapping structure within the Device Object Dictionary. If variable PDO-mapping is supported the number of PDOs and the mapping of application objects into a PDO may be transmitted to a device during the device configuration process by applying the SDO services (refer to “Service Data Objects - SDO” on page 31) to the corresponding entries of the Object Dictionary. Number and length of PDOs of a device are application specific and are specified within the device profile.

There are two ways PDOs can be used. The first is data transmission and the second data reception. It is distinguished in Transmit-PDOs (TPDOs) and Receive-PDOs (RPDOs). Devices supporting TPDOs are PDO producer and devices which are able to receive PDOs are called PDO consumer /1/.

6.1 PDO Configuration Options

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

6.1.1 COM_PDO_RCV_MAX

This symbol defines the total number of receive PDOs used by the CANopen Master stack. The number can be in the range between 0 (no receive PDO) and 512 (maximum number of PDOs). The number affects the required RAM size.

6.1.2 COM_PDO_TRM_MAX

This symbol defines the total number of transmit PDOs used by the CANopen Master stack. The number can be in the range between 0 (no transmit PDO) and 512 (maximum number of PDOs). The number affects the required RAM size.
6.2 PDO Functions

The PDO functions of the CANopen Master protocol stack have the prefix ComPdo and are located in the file `com_pdo.c` within the source directory.

<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ComPdoEnable()</td>
<td>Enable or disable a PDO</td>
</tr>
<tr>
<td>ComPdoEventReceive()</td>
<td>PDO receive event, Callback function (com_user.c)</td>
</tr>
<tr>
<td>ComPdoGetCounter()</td>
<td>Retrieve PDO counter</td>
</tr>
<tr>
<td>ComPdoGetData()</td>
<td>Get PDO data</td>
</tr>
<tr>
<td>ComPdoIdAdd()</td>
<td>Add PDO to master</td>
</tr>
<tr>
<td>ComPdoSendAsync()</td>
<td>Send asynchronous PDO</td>
</tr>
<tr>
<td>ComPdoSetData()</td>
<td>Set PDO data</td>
</tr>
</tbody>
</table>

*Table 6: PDO functions*
6.2.1 ComPdoEnable

Syntax

```c
ComStatus_tv ComPdoEnable(
    uint8_t ubNetV,
    uint16_t uwPdoV,
    uint8_t ubEnableV)
```

Function

This function enables (ubEnableV = 1) or disables (ubEnableV = 0) the PDO with index uwPdoV. A PDO is disabled by default upon creation (refer to `ComPdoIdAdd()`).

Parameters

- `ubNetV`: CANopen Network channel
- `uwPdoV`: PDO index
- `ubEnableV`: Enable/Disable PDO

Return value

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.
6.2.2 ComPdoEventReceive

Syntax

```c
void ComPdoEventReceive(
    uint8_t ubNetV,
    uint16_t uwPdoV)
```

Function

The CANopen Master PDO event receive handler will be called when a configured PDO has been received.

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

Parameters

- `ubNetV` CANopen Network channel
- `uwPdoV` PDO index

Return value

None

The following example shows a receive handler for the PDO with index number 3.

```c
//--------------------------------------------------------//
// ComPdoEventReceive()                                   //
// Function handler for PDO Receive                       //
//--------------------------------------------------------//
void ComPdoEventReceive(uint8_t ubNetV, uint16_t uwPdoV)
{
    uint8_t aubPdoDataT[8];

    if((ubNetV == eCOM_NET_1) && (uwPdoV == 3))
    {
        ComPdoGetData(ubNetV, uwPdoV, &aubPdoDataT[0]);
        // now aubPdoDataT[] contains all data of
        // received PDO
        // user can handle this data and call
        // application functions from here
    }
}
```

Example 7: PDO receive event handler
6.2.3 ComPdoGetCounter

**Syntax**

```c
ComStatus_tv ComPdoGetCounter(  
    uint8_t ubNetV,  
    uint16_t uwPdoV,  
    uint32_t * pulCounterV)
```

**Function**
The function returns the number of PDOS that have been sent (for Transmit-PDOs) or received (for Receive-PDOs) by the CANopen Master.

**Parameters**
- `ubNetV` CANopen Network channel
- `uwPdoV` PDO index
- `pulCounterV` Pointer to counter variable

**Return value**
On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.

6.2.4 ComPdoGetData

**Syntax**

```c
ComStatus_tv ComPdoGetData(  
    uint8_t ubNetV,  
    uint16_t uwPdoV,  
    uint8_t * pubDataV)
```

**Function**
The function reads 8 bytes of data from the PDO defined by the parameter `uwPdoV`. The application must ensure that `pubDataV` points to a buffer with at least 8 bytes.

**Parameters**
- `ubNetV` CANopen Network channel
- `uwPdoV` PDO index
- `pubDataV` Pointer to data buffer

**Return value**
On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.
6.2.5 ComPdoIdAdd

Syntax

```c
ComStatus_tv ComPdoIdAdd(
    uint8_t ubNetV,
    uint16_t uwPdoV,
    uint32_t ulIdV,
    uint8_t ubDlcV,
    uint8_t ubTypeV,
    uint16_t uwTimeV)
```

Function

This function adds a new PDO to the CANopen Master. The PDO type is defined by adding the definition COM_PDO_RCV for Receive-PDOs and COM_PDO_TRM for Transmit-PDOs to the parameter `uwPdoV`.

Parameters

- `ubNetV`: CANopen Network channel
- `uwPdoV`: PDO index
- `ulIdV`: PDO identifier
- `ubDlcV`: PDO DLC value
- `ubTypeV`: PDO transmission type
- `uwTimeV`: PDO timer given in ms

Return value

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.

The following code shows an example for PDO initialisation. A transmit PDO is added with an identifier value of 120h and DLC value of 3. The PDO is time triggered with a period of 5 seconds. A receive PDO is added with an identifier value of 181h. The PDO is event triggered.

```c
uint8_t aubPdoData[] = {0x12,0x34,0x56,0x78,
                         0x9A,0xBC,0x00,0x00};
//--------------------------------------------------------
// initialise transmitt PDO
// initialise transmitt PDO
ComPdoIdAdd(ubNetV, (COM_PDO_TRM | 0), 0x0120, 3, 0xFE, 5000);
ComPdoSetData(ubNetV, (COM_PDO_TRM | 0), &aubPdoData[0]);
ComPdoEnable(ubNetV, (COM_PDO_TRM | 0), 1);
//--------------------------------------------------------
// initialise receive PDO
// initialise receive PDO
ComPdoIdAdd(ubNetV, (COM_PDO_RCV | 3), 0x0181, 8, 0xFF, 0);
ComPdoEnable(ubNetV,(COM_PDO_RCV | 3), 1);
```

Example 8: PDO initialisation
### 6.2.6 ComPdoSendAsync

**Syntax**

```c
ComStatus_tv ComPdoSendAsync(
    uint8_t    ubNetV,
    uint16_t   uwPdoV)
```

**Parameters**

- `ubNetV` : CANopen Network channel
- `uwPdoV` : PDO index

**Return value**

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.

**Function**

This function sends a PDO defined by the index `uwPdoV` by the application program (asynchronous). The PDO must have been configured (see `ComPdoIdAdd()`) and enabled (see `ComPdoEnable()`) in advance.

```c
uint8_t aubPdoData[] = {0x12,0x34,0x56,0x78,
    0x9A,0xBC,0x00,0x00};

//--------------------------------------------------------
// initialise transmitt PDO
//
// ComPdoIdAdd(ubNetV, (COM_PDO_TRM | 0), 0x0120, 3, 0xFE,
//                5000);
ComPdoSetData(ubNetV, (COM_PDO_TRM | 0), &aubPdoData[0]);
ComPdoEnable(ubNetV, (COM_PDO_TRM | 0), 1);
ComPdoSendAsync(ubNetV,(COM_PDO_TRM | 0));
```

*Example 9: PDO transmission by application*
### 6.2.7 ComPdoSetData

**Syntax**
```
ComStatus_t ComPdoSetData(
    uint8_t ubNetV,
    uint16_t uwPdoV,
    uint8_t * pubDataV)
```

**Function**
The function sets the data bytes of a PDO. The application must ensure that `pubDataV` points to a buffer with at least 8 bytes. The PDO must have been configured (see `ComPdoIdAdd()`) in advance.

**Parameters**
- `ubNetV` : CANopen Network channel
- `uwPdoV` : PDO index
- `pubDataV` : Pointer to data buffer

**Return value**
On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.
7. Emergency Service - EMCY

Emergency objects are triggered if device internal error situations occur 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/.

7.1 EMCY Producer Configuration Options

The file `com_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.

7.1.1 COM_EMCY_SUPPORT

The symbol defines if the EMCY service is supported or not. It is enabled by default. Setting this symbol to 0 disables the support of EMCY producer functionality by the CANopen Master Stack.

7.1.2 COM_EMCY_QUEUE

The symbol defines the size of the internal EMCY message queue. If the queue size is not sufficient, subsequent calls of `ComEmcySend()` may fail.

7.1.3 COM_BUS_EMCY

With this symbol it is possible to enable the transmission of emergency messages upon CAN bus errors. It is disabled by default. Setting this symbol to 1 enables transmission of EMCY messages upon CAN bus errors.
7.2 EMCY Producer Functions

The EMCY producer functions of the CANopen Master protocol stack have the prefix ComEmcy and are located in the file com_emcy.c within the source directory.

<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ComEmcyEnable()</td>
<td>Enable or disable EMCY producer</td>
</tr>
<tr>
<td>ComEmcySend()</td>
<td>Send an EMCY message</td>
</tr>
<tr>
<td>ComEmcySetId()</td>
<td>Set identifier of EMCY message</td>
</tr>
</tbody>
</table>

*Table 7: EMCY Producer Service functions*
### 7.2.1 ComEmcyEnable

**Syntax**

```c
ComStatus_tv ComEmcyEnable(
    uint8_t   ubNetV,
    uint8_t   ubEnableV)
```

**Function**

This function enables (ubEnableV = 1) or disables (ubEnableV = 0) the EMCY service. The EMCY service is enabled by default.

**Parameters**

- **ubNetV** CANopen Network channel
- **ubEnableV** Enable/Disable EMCY producer service

**Return value**

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.

### 7.2.2 ComEmcySend

**Syntax**

```c
ComStatus_tv ComEmcySend(
    uint8_t   ubNetV,
    uint16_t  uwEmcyCodeV,
    uint8_t * pubManCodeV)
```

**Function**

This function sends an EMCY message with the code `uwEmcyCodeV` and an optional manufacturer data of 5 bytes.

**Parameters**

- **ubNetV** CANopen Network channel
- **uwEmcyCodeV** Emergency code
- **pubManCodeV** Pointer to manufacturer data

**Return value**

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.
7.2.3 ComEmcySetId

**Syntax**

```c
ComStatus_tv ComEmcySetId(
    uint8_t ubNetV,
    uint32_t ulIdV)
```

**Function**

This function changes the identifier value for the EMCY service.

**Parameters**

- `ubNetV` CANopen Network channel
- `ulIdV` Identifier value

**Return value**

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.
7.3 EMCY Consumer Configuration Options

The file `com_conf.h` holds definitions for the configuration of the EMCY consumer 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.

7.3.1 COM_EMCY_CONS_SUPPORT

The symbol defines if the EMCY Consumer service is supported. If a consumer EMCY was received the `ComEmcyConsEventReceive()` will be called and the application can handle the event. This symbol can only be set to 1 if `COM_EMCY_SUPPORT` is enabled. The service is disabled by default.
7.4 EMCY Consumer Functions

The EMCY consumer functions of the CANopen Master protocol stack have the prefix ComEmcyCons and are located in the file com_emcy.c within the source directory.

<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ComEmcyConsEnable()</td>
<td>Enable EMCY consumer</td>
</tr>
<tr>
<td>ComEmcyConsEventReceive()</td>
<td>EMCY receive event, Callback function (com_user.c)</td>
</tr>
<tr>
<td>ComEmcyConsGetData()</td>
<td>Get EMCY data</td>
</tr>
<tr>
<td>ComEmcyConsSetId()</td>
<td>Set EMCY consumer identifier value</td>
</tr>
</tbody>
</table>

*Table 8: EMCY consumer functions overview*
7.4.1 ComEmcyConsEnable

**Syntax**

```c
ComStatus_tv ComEmcyConsEnable(
    uint8_t ubNetV,
    uint8_t ubNodeIdV,
    uint8_t ubEnableV)
```

**Function**

This function enables \(ubEnableV = 1\) or disables \(ubEnableV = 0\) the EMCY consumer service for a CANopen device with node-ID value `ubNodeIdV`. The value range for `ubNodeIdV` is limited by the value of `COM_NODE_MAX`.

**Parameters**

- `ubNetV` CANopen Network channel
- `ubNodeIdV` Node-ID value
- `ubEnableV` Enable/Disable EMCY consumer service

**Return value**

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.

7.4.2 ComEmcyConsEventReceive

**Syntax**

```c
void ComEmcyConsEventReceive(
    uint8_t ubNetV,
    uint8_t ubNodeIdV)
```

**Function**

This event handler will be called on reception of an EMCY message. The value range for `ubNodeIdV` is limited by the value of `COM_NODE_MAX`.

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

**Parameters**

- `ubNetV` CANopen Network channel
- `ubNodeIdV` Node-ID value

**Return value**

None
### 7.4.3 ComEmcyConsGetData

**Syntax**

```c
ComStatus_tv ComEmcyConsGetData(
    uint8_t ubNetV,
    uint8_t ubNodeIdV,
    uint8_t * pubDataV)
```

**Function**

This function reads 8 bytes of data from the EMCY message defined by the parameter `ubNodeIdV`. The application must ensure that the pointer `pubDataV` points to a buffer with at least 8 bytes. The value range for `ubNodeIdV` is limited by the value of `COM_NODE_MAX`.

**Parameters**

- `ubNetV`: CANopen Network channel
- `ubNodeIdV`: Node-ID value
- `pubDataV`: Pointer to data destination

**Return value**

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.
7.4.4 ComEmcyConsSetId

**Syntax**
```
ComStatus_tv ComEmcyConsSetId(
    uint8_t  ubNetV,
    uint8_t  ubNodeIdV,
    uint32_t ulIdV)
```

**Function**
This function sets EMCY Identifier `ulIdV` for a CANopen device with the node-ID value of `ubNodeIdV`. The value range for `ubNodeIdV` is limited by the value of `COM_NODE_MAX`.

**Parameters**
- `ubNetV`: CANopen Network channel
- `ubNodeIdV`: Node Identifier of CANopen device
- `ulIdV`: Identifier value for Emcy

**Return value**
On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.
8. Synchronisation Service - SYNC

The Synchronisation Object is broadcasted periodically by the SYNC producer. This SYNC provides the basic network clock. The time period between the SYNCs is specified by the standard parameter communication cycle period (Object 1006h: Communication Cycle Period), which may be written by a configuration tool to the application devices during the boot-up process.

There can be a time jitter in transmission by the SYNC producer corresponding approximately to the latency due to some other message being transmitted just before the SYNC. In order to guarantee timely access to the CAN bus the SYNC is given a very high priority identifier (1005h).

Devices which operate synchronously may use the SYNC object to synchronise their own timing with that of the Synchronisation Object producer. The details of this synchronisation are device dependent /1/.

8.1 SYNC Configuration Options

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

8.1.1 COM_SYNC_SUPPORT

The Synchronisation Object is broadcasted periodically by the SYNC producer. This SYNC provides the basic network clock. This switch is used to enable or disable the SYNC capability. The service is enabled by default.
8.2 SYNC Functions

The SYNC functions of the CANopen Master protocol stack have the prefix ComSync and are located in the file com_sync.c within the source directory.

<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ComSyncEnable()</td>
<td>Enable SYNC message</td>
</tr>
<tr>
<td>ComSyncSetCounter()</td>
<td>Set SYNC counter</td>
</tr>
<tr>
<td>ComSyncSetCycleTime()</td>
<td>Set SYNC cycle time</td>
</tr>
<tr>
<td>ComSyncSetId()</td>
<td>Set SYNC identifier</td>
</tr>
</tbody>
</table>

Table 9: SYNC Service functions

The generation of SYNC messages is disabled by default. The SYNC service identifier is set to 080h by default.

The following code example shows how to change the default identifier to 181h and to generate SYNC message with a cycle time of 10ms.

```
ComSyncSetId(eCOM_NET_1, 0x181);       // set new identifier
ComSyncSetCycleTime(eCOM_NET_1, 10000); // set 10 ms cycle time
ComSyncEnable(eCOM_NET_1, 1);          // start SYNC service
```

Example 10: Generating a SYNC message
8.2.1 ComSyncEnable

**Syntax**

```c
ComStatus_tv ComSyncEnable(
    uint8_t ubNetV,
    uint8_t ubEnableV)
```

**Function**

This function enables (ubEnableV = 1) or disables (ubEnableV = 0) the SYNC service. The SYNC service is disabled by default.

**Parameters**

- **ubNetV**: CANopen Network channel
- **ubEnableV**: Enable / Disable SYNC service

**Return value**

On success the value eCOM_ERR_OK is returned. On failure a negative value from the ComErr_e enumeration is returned.

8.2.2 ComSyncSetCounter

**Syntax**

```c
ComStatus_tv ComSyncSetCounter(
    uint8_t ubNetV,
    uint8_t ubCounterV)
```

**Function**

This function changes the counter value for the SYNC service. Valid values for `ubCounterV` are 0 and 2 to 240. A value of 0 sets the DLC of the SYNC service to 0 (default).

The following example shows how the set the counter to a maximum (overflow) value of 10. Please note that the SYNC counter value always starts with 1.

```c
ComSyncSetCycleTime(eCOM_NET_1, 0);    // stop SYNC
ComSyncSetCounter(eCOM_NET_1, 10);     // set SYNC counter max value to 10
ComSyncSetCycleTime(eCOM_NET_1, 10000); // start SYNC with 10 ms cycle time
```

*Example 11: Using SYNC counter*

**Parameters**

- **ubNetV**: CANopen Network channel
- **ubCounterV**: Highest counter value

**Return value**

On success the value eCOM_ERR_OK is returned. On failure a negative value from the ComErr_e enumeration is returned.
8.2.3 ComSyncSetCycleTime

**Syntax**

```c
ComStatus_tv ComSyncSetCycleTime(
    uint8_t ubNetV,
    uint32_t ulTimeV)
```

**Function**

This function changes the cycle time for the SYNC service. The parameter `ulTimeV` is passed as a multiple of 1 microsecond.

**Parameters**

- `ubNetV` CANopen Network channel
- `ulTimeV` SYNC cycle time

**Return value**

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.

8.2.4 ComSyncSetId

**Syntax**

```c
ComStatus_tv ComSyncSetId(
    uint8_t ubNetV,
    uint32_t ulIdV)
```

**Function**

This function changes the identifier value for the SYNC service. The SYNC service must be disabled (refer to “ComSyncEnable” on page 61) prior to changing the identifier.

**Parameters**

- `ubNetV` CANopen Network channel
- `ulIdV` Identifier value

**Return value**

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.
9. 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 (baud rate)
- LSS address (Identity Object, Index 1018h)

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

9.1 LSS Configuration Options

The file `com_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.

9.1.1 COM_LSS_SUPPORT

This symbol is used to enable or disable the LSS functionality. The service is enabled by default.
9.2 LSS Functions

The LSS functions of the CANopen Master protocol stack have the prefix `ComLss` and are located in the file `com_lss.c` within the source directory.

<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ComLssConfigureBitTiming()</code></td>
<td>Configure bit-timing</td>
</tr>
<tr>
<td><code>ComLssConfigureNodeId()</code></td>
<td>Configure node-ID</td>
</tr>
<tr>
<td><code>ComLssGetStatus()</code></td>
<td>Get status of LSS master state machine</td>
</tr>
<tr>
<td><code>ComLssInquiryService()</code></td>
<td>inquires LSS information</td>
</tr>
<tr>
<td><code>ComLssSetTimeout()</code></td>
<td>Set communication timeout</td>
</tr>
<tr>
<td><code>ComLssStoreConfiguration()</code></td>
<td>Store LSS configuration</td>
</tr>
<tr>
<td><code>ComLssSwitchModeGlobal()</code></td>
<td>Switch mode global</td>
</tr>
<tr>
<td><code>ComLssSwitchModeSelective()</code></td>
<td>Switch mode selective</td>
</tr>
</tbody>
</table>

*Table 10: LSS functions overview*
9.2.1 ComLssConfigureBitTiming

Syntax

```c
ComStatus_tv ComLssConfigureBitTiming(
    uint8_t ubNetV,
    uint8_t ubBittimeSelV)
```

Function

By means of the Configure Bit Timing Parameters service the LSS Master sets the new bit-timing on a LSS Slave.

Parameters

- `ubNetV` CANopen Network channel
- `ubBittimeSelV` New bit-timing value

Return value

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.

9.2.2 ComLssConfigureNodeId

Syntax

```c
ComStatus_tv ComLssConfigureNodeId(
    uint8_t ubNetV,
    uint8_t ubNodeIdV)
```

Function

By means of this service the LSS Master configures the NMT-address parameter of a LSS Slave. This service allows only one LSS Slave in configuration mode at the same time. The remote result parameter confirms the success or failure of the service. In case of a failure the reason may optionally be confirmed.

Parameters

- `ubNetV` CANopen Network channel
- `ubNodeIdV` New Node-ID value

Return value

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.
### 9.2.3 ComLssGetStatus

**Syntax**

```c
ComStatus_tv ComLssGetStatus(
    uint8_t ubNetV,
    uint8_t * pubStatusV)
```

**Function**

This function returns the status of the LSS master state machine. Valid codes passed the pointer `pubStatusV` are given in the enumeration `ComLssStat_e`.

**Parameters**

- `ubNetV` CANopen Network channel
- `pubStatusV` Pointer to status variable

**Return value**

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.

### 9.2.4 ComLssInquiryService

**Syntax**

```c
ComStatus_tv ComLssInquiryService(
    uint8_t ubNetV,
    uint8_t ubServiceV)
```

**Function**

The inquiry services are available only in configuration mode. Valid service types passed `pubStatusV` are given in the enumeration `ComLssInqSrv_e`.

**Parameters**

- `ubNetV` CANopen Network channel
- `ubServiceV` Service type

**Return value**

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.
9.2.5 ComLssSetTimeout

Syntax

```c
void ComLssSetTimeout(
    uint8_t ubNetV,
    uint16_t uwTimeV)
```

Function

This function sets a timeout value for a LSS communication. If the response to a LSS request by the master takes longer than this timeout value, the state of the communication will be changed to `eCOM_LSS_STAT_TIMEOUT`.

The state of the LSS communication can be evaluated by calling `ComLssGetStatus()`. Please note that the timeout value is a multiple of the timer period (`COM_TIMER_PERIOD`).

A value supplied to this function will be rounded towards the next lower value if necessary.

Parameters

- `ubNetV` CANopen Network channel
- `uwTimeV` Timeout value in milliseconds

Return value

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.

9.2.6 ComLssStoreConfiguration

Syntax

```c
void ComLssStoreConfiguration(
    uint8_t ubNetV)
```

Function

This function is used to store the configured parameters.

Parameters

- `ubNetV` CANopen Network channel

Return value

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.
9.2.7 ComLssSwitchModeGlobal

Syntax

```c
ComStatus_tv ComLssSwitchModeGlobal(  
    uint8_t ubNetV,  
    uint8_t ubModeV)
```

Function

The Switch Mode Services control the mode attribute of a LSS Slave. LSS provides two ways to put a LSS Slave device into configuration mode, Switch Mode Global and Switch Mode Selective.

Switch Mode Selective switches exactly one LSS Slave device into configuration mode (refer to "ComLssSwitchModeSelective" on page 69).

Switch Mode Global switches all LSS Slaves between configuration and operation mode. LSS Slaves will be set into configuration mode when `ubModeV = 1` and into operation mode when `ubModeV = 0`.

Parameters

- `ubNetV` CANopen Network channel
- `ubModeV` LSS mode

Return value

On success the value `eCOM_ERR_OK` is returned. On failure a negative value from the `ComErr_e` enumeration is returned.
9.2.8 ComLssSwitchModeSelective

Syntax

```c
ComStatus_tv ComLssSwitchModeSelective(
    uint8_t ubNetV,
    uint32_t ulVendorIdV,
    uint32_t ulProdCodeV,
    uint32_t ulRevisionNumV,
    uint32_t ulSerialNumV)
```

Function

The Switch Mode Services control the mode attribute of a LSS Slave. LSS provides two ways to put a LSS Slave into configuration mode, Switch Mode Global and Switch Mode Selective.

Switch Mode Selective switches exactly one LSS Slave device into configuration mode.

Switch Mode Global switches all LSS Slaves between configuration and operation mode (refer to "ComLssSwitchModeGlobal" on page 68).

Parameters

- **ubNetV**: CANopen Network channel
- **ulVendorIdV**: Vendor ID (1018h, sub 1)
- **ulProdCodeV**: Product code (1018h, sub 2)
- **ulRevisionNumV**: Revision number (1018h, sub 3)
- **ulSerialNumV**: Serial number (1018h, sub 4)

Return value

On success the value **eCOM_ERR_OK** is returned. On failure a negative value from the **ComErr_e** enumeration is returned.
A Source Code Agreement

This source code agreement (hereinafter - "Agreement") is made between MicroControl GmbH & Co. KG, Lindlaufstr. 2c, 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.
Source Code Agreement

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

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 Confi-
dential 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 pur-
poses of this Agreement. Notwithstanding the foregoing, either party may use or disclose Confi-
dential 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 pro-
tecting 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.

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) consti-
tutes a valuable asset of MicroControl and therefore agrees that only the following Licensee em-
ployees 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 provi-
sions 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 em-
ployees, agents and contractors allowed access to the Confidential Information to satisfy its obliga-
tions 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.
Source Code Agreement

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.
B. Index

E
EMCY 49
EMCY Consumer Functions 54
  ComEmcyConsGetData 56
  ComEmcyConsSetId 57
EMCY Producer Functions 50
  ComEmcyEnable 51
  ComEmcySend 51
  ComEmcySetId 52

L
LSS 63
LSS Functions 64
  ComLssConfigureBitTiming 65
  ComLssConfigureNodeId 65
  ComLssGetStatus 66
  ComLssInquiryService 66
  ComLssSetTimeout 67
  ComLssStoreConfiguration 67
  ComLssSwitchModeGlobal 68
  ComLssSwitchModeSelective 69

M
Manager 15
Manager Functions 18
  ComMgrEventBus 20
  ComMgrInit 19
  ComMgrMasterInit 20
  ComMgrRelease 21
  ComMgrStart 21
  ComMgrStop 22
  ComMgrTimerEvent 23

N
NMT 25
NMT Functions 26
  ComNmtEventHeartbeat 27
  ComNmtEventIdCollision 27
  ComNmtEventStateChange 28
  ComNmtSetHbConsTime 29
  ComNmtSetHbProdTime 29
  ComNmtSetNodeState 28, 30

P
PDO 41
PDO Functions 42
  ComPdoEventReceive 44
  ComPdoGetCounter 45
  ComPdoGetData 45
  ComPdoIdAdd 46
ComPdoSendAsync 47
ComPdoSetData 48

S

SDO 31
SDO Client Functions 32
  ComSdoGetAbortCode 33
  ComSdoGetStatus 33
  ComSdoReadData 34
  ComSdoReadObject 35
  ComSdoSetConfiguration 36
  ComSdoSetTimeout 36
  ComSdoWriteData 37
  ComSdoWriteObject 38

SYNC 59

SYNC Functions 60
  ComSyncEnable 61
  ComSyncSetCounter 61
  ComSyncSetCycleTime 62
  ComSyncSetId 62
MicroControl reserves the right to modify this manual and/or product described herein without further notice. Nothing in this manual, nor in any of the data sheets and other supporting documentation, shall be interpreted as conveying an express or implied warranty, representation, or guarantee regarding the suitability of the products for any particular purpose. MicroControl does not assume any liability or obligation for damages, actual or otherwise of any kind arising out of the application, use of the products or manuals.

The products described in this manual are not designed, intended, or authorized for use as components in systems intended to support or sustain life, or any other application in which failure of the product could create a situation where personal injury or death may occur.

© 2013 MicroControl GmbH & Co. KG, Troisdorf