МАРАФОН
CAN технология

CANopen Library

Programmer's guide

Project code **0001**$_h$

Mosfilmovskaya Street 17B, 117330 Moscow, the Russian Federation.
Phone: +7-495-9391324 Fax: +7-495-9395659
Email: info@marathon.ru **www.marathon.ru**

Table of Contents

Basic features

The CANopen library enables you to develop master and slave devices compliant to CiA 301 v. 4.2. The library supports LSS slave, based on the CiA DSP 305 v. 2.2 specification. The library code is written in ANSI C language taking into account scalability and portability on different platforms.

To access the network at the data link layer the library uses the CHAI driver API. All dependencies of the runtime environment are available in separate modules. Thus, the source code of the library itself is not dependent on specific platform and the same for embedded applications and for tasks that are running under operating systems: Windows, Linux and others.

- The library provides hard real-time operation mode. Its architecture is based on re–entrant functions that allow asynchronous call from the application program.
- Communication profile objects perform a full reconfiguration in accordance with CiA 301.
- Initialization of communication objects is performed according to the generic pre-defined connection set.

The functionality of the library

- CANopen SDO protocol is supported in all defined by the CiA 301 modes: expedited, segmented and block.
- All PDO transmission types are supported (cyclic, acyclic, synchronous, asynchronous, RTR only). Both static and dynamic PDO mapping can be used.
- SYNC protocol operates with or without the SYNC counter.
- All NMT services and NMT protocols are supported.
- Full LSS protocols family is implemented, including Fastscan.

The library limitations

- The maximum size of any object should not exceed $7FFFFFFF_h$ (2147483647) bytes.
- The minimum value of the CANopen timer period is 100 microseconds (frequency not exceeding 10 KHz).
- The CANopen communication profile supports only 11-bit CAN identifiers. 29-bit CAN-IDs are reserved and are not used in the CANopen protocol. The library ignores all incoming frames with 29-bit identifiers.

Optimization of the library source code

Using the library it is strongly recommended to disable algorithmic optimization of the source code by a compiler. Optimization often violates compliance of the algorithms written in high level language and machine code generated by the compiler. Partial suppression of the optimization techniques, for example, additional variable declarations "volatile", do not guarantee that all errors and side effects of the optimization will be eliminated.

Documentation

The CANopen library is developed based on the CAN in Automation specifications:

CiA 301	v. 4.2	CANopen application layer and communication profile.
CiA 303 p. 3	v. 1.4	Indicator specification.
CiA 305	v. 2.2	Layer setting services.
CiA 306	v. 1.3	Electronic data sheet specification.

A supplement to this manual are the guides:
"Adapted slave for Windows OS";
"Adapted master for Windows OS";
"DLL master for Windows OS with LabVIEW application".

Abbreviations and definitions

CiA	CAN in Automation http://www.can-cia.org/
CAN-ID	CAN data link identifier.
COB-ID	CANopen communication object identifier.
NMT	Network management.
LSS	Layer setting services.
EDS	Electronic data sheet.
DCF	Device configuration file.
PDO	Process data object.
RTR	Remote transmission request.
SDO	Service data object.
M	Mandatory object.
O	Optional object.
LSB	Least significant bit/byte.
MSB	Most significant bit/byte.
RO	Read only access.
WO	Write only access.
RW	Read and write access.
RWR	Read / write on process input (TPDO).
RWW	Read / write on process output (RPDO).

Basic data types

boolean	Logical value TRUE or FALSE.
int8	Signed integer 8 bit.
unsigned8	Unsigned integer 8 bit.
int16	Signed integer 16 bit.

unsigned16	Unsigned integer 16 bit.
int32	Signed integer 32 bit.
unsigned32	Unsigned integer 32 bit.
int64	Signed integer 64 bit.
unsigned64	Unsigned integer 54 bit.
real32	32 bit floating point.
real64	64 bit floating point.
vis–string	Visible string (values 0 and 20_h to $7E_h$).
octet-string	Octet string (values 0 to FF_h).

Revision history

Version 2.3

CANopen master for Windows OS, implemented as a DLL module is included in the library. One of the master applications is an interface module with the LabVIEW package.
The library is adapted for passing CANopen conformance test of the third major version.

Version management

The library supports simple version control system based on the C language pre-processor directives. Each library module is enclosed in the conditional macro:

```
#if CHECK_VERSION(2, 3, 0)
        library module source code
#endif
```

The first argument of the macro means the major version of the library, the second is the minor version and the third is the issue number. All library modules must have the same major and minor versions, and the issue number should not be below the minimum (usually zero).
If the major and minor versions are changed, the following rules apply:

- if the module code is not changed, it is assigned a zero issue number;
- if the module code is updated, when changing the major or minor versions, it is assigned the first issue number.

Version history of the library module is saved by storing its latest release as C language comment. For example, the set of macros

```
#if CHECK_VERSION(2, 2, 0)
//  CHECK_VERSION(2, 1, 0)
//  CHECK_VERSION(2, 0, 0)
//  CHECK_VERSION(1, 7, 1)
```

means that the current 2.2.0 version is identical to the version 1.7.1 of this module.

Assembly and installation of the library

Build and installation of the library for operating systems Linux and Windows.

CAN driver installation

Install CAN data link layer driver CHAI in accordance with the instructions posted on the website http://can.marathon.ru/page/prog/chai.
Note. The library assembly in the test mode does not require CAN controller and the driver.

The library layout

The CANopen directory contains the sub-directory of the type 2.3.x, which identifies the library version. The first number means the major version of the library, the second is the minor version. All library modules must have the same major and minor versions, and the issue number should not be below the minimum (usually zero). The specified sub-directory, in turn, contains the following three directories:

- Linux – contains modules for Linux operating system.
- src – "root" directory of the CANopen source code library. Location of all modules is given relative to this directory.
- win – here are placed the project files (*.sln, *.suo, *.vcxproj, *.vcxproj.*) for the Microsoft Visual C++ 2010. These projects can be used to build target application based on the CANopen library.

Windows operational system

To build the CANopen application in the header file \include__can_defines.h you should choose the type of operating system Windows: #define CAN_OS_WIN32 and set the build mode of the target application CAN_APPLICATION_MODE. You can override other configuration settings, if necessary.
To compile the application using Microsoft Visual C++ 2010, you must perform the following operations:

- Specify the directories where the header files of the library and CHAI driver are located. For example, ..\src\include for the CANopen library files and C:\Program Files (x86)\CHAI – 2.10.4\include for the CHAI driver header files. Navigation: Project –> Properties –> Configuration properties –> C/C++ –> Additional Include Directories.
- Specify the directories where CHAI driver lib file is located. For instance, C:\Program Files (x86)\CHAI – 2.10.4\lib. Navigation: Project –> Properties –> Configuration properties –> Linker –> Additional Library Directories.
- Build target application. Navigation: Build –> Build Solution.

Linux operational system

To build the CANopen application in the header file \include__can_defines.h you should choose the type of operating system Linux: #define CAN_OS_LINUX and set the build mode of the target application CAN_APPLICATION_MODE. You can override other configuration settings, if necessary.
To compile the application, run make command with one of the options: 'make canmaster' for master application, 'make canslave' for the slave one or 'make cantest' to compile the test

application. In the Make.vars file you may need to adjust the path to the header and library files of the CHAI driver. As a result of the compilation the executable application module *canapp is generated.

Implementation technology of the library functions and protocols

CAN controller acceptance filter

Incoming CAN frames filtering is carried out using the CAN–ID bit mask. The filter passes only those frames in which some bits have a certain fixed value. Since all CAN nodes must receive NMT frames with zero CAN-ID, the filter should pass all the CAN–IDs in which the value of any bit is equal to zero. Thus, single-level filtering can't get rid of the frames, where the bits of the identifier can accept both 0 and 1 values. That is, the efficiency of this filtration depends on the CAN node-ID. The node number 127 will receive all CAN frames, because it needs to handle NMT requests with the identifier equal to zero, as well as frames addressed to the node with the value of seven least significant CAN–ID bits equal to 1 (the node number for the generic pre-defined connection set). Two-level filter does not have this drawback. Here it is possible to filter separately the broadcast frames with zero value of the node number field (NMT, SYNC, TIME STAMP) and with the value of the seven least significant CAN–ID bits corresponding to the number of the node. Thus, for the generic pre-defined connection set only the frames destined to this CAN node are selected.

Incoming frame CAN–IDs processing technique

The library supports two ways of incoming frames CAN-IDs processing: dynamic and static. The dynamic method requires significantly less memory, but not as computing speed effective as static. For the dynamic method the disordered array of records is created. It contains the value of CAN–ID and the corresponding communication object dictionary index. When receiving a CAN frame linear search of the index, corresponding to the received CAN-ID, is performed. Total number of CAN-IDs, processed by the dynamic method, is determined by the parameters CAN_NOF_RECVCANID_MASTER (for the master) and CAN_NOF_RECVCANID_SLAVE (for the slave) in the header file \include__can_defines.h. The dynamic method is effective when the total number of processed CAN–IDs do not exceed 50..100, depending on the CPU performance. The static method is only used for the CANopen master and 11-bit CAN–IDs. It creates an array which size corresponds to the maximum possible number of identifiers that are mapped to the communication object dictionary indexes. The CAN–IDs processing method for the CANopen master is defined by the parameter CAN_MASTER_RECVCANID_METHOD in the header file \include__can_defines.h. CAN controller acceptance filter is set only for the dynamic method. For CANopen slave only the dynamic method is used, due to small number of configurable CAN-IDs (usually 11).

Restricted CAN identifiers

From the entire set of restricted CAN-IDs, NMT object identifiers are processed regardless of the settings: NMT (CAN–ID = 0_h) and NMT Error Control (CAN–IDs 701_h to $77F_h$). In addition, when LSS protocol is activated identifiers $7E5_h$ (transmitted by the LSS master) and $7E4_h$ (transmitted by the LSS slave) are also handled directly. Assignment of these identifiers to other objects will not allow the latter to use them, since the corresponding CAN–IDs are intercepted before processing any configurable identifiers. Restricted CAN-IDs are also checked and banned in any configurable COB-IDs. For SDO objects CAN-ID values can only be in acceptable ranges. However, the library does not exercise control over the use of other configurable identifiers.

Object dictionary implementation

Object dictionary entries are implemented statically. This enables asynchronous access to the dictionary, for example, with the PDOs. Object dictionary implementation example for the slave test profile is given in the \slave__obdms_slave_test.h module.
As a possible access method to the slave devices object dictionary, the library offers mapping in the master slaves application profile object dictionary entries. This enables direct master-slave data exchange with both SDO and PDO protocols. Communication profile area of the master (client) and slaves (servers) do not require mapping and implemented independently. An example of the object dictionary mapping implementation for the slave test profile is given in the module \master__obdms_master_test.h. Another examples of the object dictionary implementation and operations can be found in adapted master and slave library versions, as well as in the CANopen DLL master.

SDO protocol implementation

Up to four data bytes, enclosed in a single CAN frame, are transmitted in an expedited SDO protocol: additional buffering in this case is not required. Segmented SDO protocol exchanges data with buffering on both the transmitting and the receiving side. During the initialization of the segmented protocol the transmitting party reads the corresponding object dictionary entry into a dynamic buffer, and the receiving party allocates a buffer of the required size. Data is updated in the object dictionary of the receiving party only after the successful completion of the entire transmission cycle. The maximum size of the object dictionary entry, transferred using the segmented SDO protocol is defined by the parameter CAN_SIZE_MAXSDOMEM. This parameter is used by the specialized signal-safe dynamic memory allocation function to determine the maximum buffer size. Block SDO protocol uses buffering on the server side and only if all the data can be placed in a dynamic buffer. But, as a rule, block data transfer is performed directly between the object dictionary entries of the transmitting and receiving parties. This is provided by an access to the relevant entries using a byte pointer. Block protocol ensures the received data consistency only after the successful completion of the entire data exchange cycle. Otherwise, the corresponding object dictionary entry should not be used.

LSS protocol implementation

LSS protocol is activated when the device detects that it has CAN node-ID equal to 255 (non-configured CANopen device). In this case, it switches to the LSS waiting state. LSS slave device becomes configured after storage the CAN node-ID value in the range 1 to 127 in the non-volatile memory. "Store configuration" protocol is utilized to fulfill the operation. LSS Fastscan protocol is active only for LSS slave devices with CAN node-ID equal to 255 (non-configured CANopen device).
LSS protocol identifiers are processed regardless of any other CANopen communication objects CAN-IDs configuration. In addition, the configured LSS device fulfills two NMT command: Reset Node and Reset Communication. After the commands execution the device goes into pre-operational NMT state. No other CANopen communication objects are used in the LSS device. However, for localization of possible application errors initialization of the CANopen communication objects is performed with the value of CAN node-ID equal to zero.

Non-volatile memory storage module

In non-volatile memory support module can_obj_re_store.c actual storage of parameters is done in ordinary static arrays of data. The consistency of data is controlled by a 16–bit CRC code. When migrating the program module to the micro-controller platform, these arrays should be replaced by the corresponding addresses of the non-volatile memory (flash, EEPROM). In addition, this platform API should be used.

Store/restore objects (1010_h, 1011_h) support 6 sub-indexes.

1010_h $sub1_h$: Save all parameters.

1010_h $sub2_h$: Save communication parameters.

1010_h $sub3_h$: Save application parameters.

1010_h $sub4_h$: Do not save any parameters.

1010_h $sub5_h$: Save CAN node-ID.

1010_h $sub6_h$: Save CAN bit rate index.

1011_h $sub1_h$: Restore all parameters.

1011_h $sub2_h$: Restore communication default parameters.

1011_h $sub3_h$: Restore application default parameters.

$1011_h sub4_h$: Restore default parameters: 1005_h, 1012_h, 1014_h, $1400_h sub1_h$, $1401_h sub1_h$, $1402_h sub1_h$, $1403_h sub1_h$, $1800_h sub1_h$, $1801_h sub1_h$, $1802_h sub1_h$, $1803_h sub1_h$. Default values of the parameters specify generic pre-defined connection set CAN-IDs, taking into account the CANopen device node-ID.

1011_h $sub5_h$: Restore CAN node-ID default selection.

1011_h $sub6_h$: Restore CAN bit rate index default selection.

CANopen library API data types and structures

Library data types

Name	Data type	Comments
canbyte	**unsigned8**	Unsigned integer 8 bit.
cannode	**unsigned8**	Unsigned integer 8 bit, CAN node-ID.
canindex	**unsigned16**	Unsigned integer 16 bit, object dictionary index.
cansubind	**unsigned8**	Unsigned integer 8 bit, object dictionary sub-index.
canlink	**unsigned16**	Unsigned integer 16 bit, 11-bit CAN–ID.
canlink	**unsigned32**	Unsigned integer 32 bit, 29-bit CAN–ID. Not used in CANopen.

Data link layer driver data types

Name	Comments
_u8	Unsigned integer 8 bit.
_s8	Signed integer 8 bit.
_u16	Unsigned integer 16 bit.
_s16	Signed integer 16 bit.
_u32	Unsigned integer 32 bit.
_s32	Signed integer 32 bit.

Data structures

Auxiliary data structures

struct sdoixs {
    canindex sdoind communication SDO parameter index (objects 1200_h to $12FF_h$).
    canindex index application object index.
    cansubind subind application object sub-index.
};
sdoixs structure defines SDO protocol communication object, as well as index and sub-index of the application object (SDO protocol multiplexer).

struct sdostatus {
    int16 state status during and after SDO client transaction.
    unsigned32 abortcode SDO abort code, if **state** has the value CAN_TRANSTATE_SDO_SRVABORT.
};
sdostatus structure places information about the status of the SDO client transaction.

API data structures

struct sdocltappl {

unsigned8 operation	basic SDO transfer mode (upload / download).
cannode node	SDO server node-ID.
unsigned32 datasize	the size of data in bytes.
canbyte *datapnt	byte pointer to the local buffer.
struct sdoixs si	SDO indexes structure.
struct sdostatus ss	SDO transaction status.

};

sdocltappl structure is used by the SDO client when exchanging data with SDO protocol.

struct canframe {

unsigned32 id	CAN–ID.
unsigned8 data[8]	CAN frame data field.
unsigned8 len	data length (0 to 8).
unsigned16 flg	CAN frame flags: bit 0 RTR, bit 2 EFF.
unsigned32 ts	CAN frame time stamp (microseconds).

};

canframe structure places CAN data link layer frame. It is defined in the CAN driver header file (**canmsg_t** structure).

Library modules placement

Library modules placement is given relative to the "root" CANopen directory.

master directory. CANopen master and SDO client modules.

- can_client.c – complete SDO client transactions support.
  can_clt_block.h – block protocol SDO client transactions.
- can_cltrans.c – basic SDO client transactions support (client request, server response).
- can_obdclt.c – access manager to the client (master) object dictionaries.
- can_obdsdo_client.c – client SDO parameters object dictionary.
- can_test_driver.c – loopback test mode CAN driver.

The following modules in this directory may be edited by the user:

- __can_test_application.c – client operations and test device object dictionary mapping.
- __obd_mans_master.c – access manager to the slave devices object dictionary mapping.
  __obdms_master_*.h – slave devices object dictionary mapping.

slave directory. CANopen slave and SDO server modules.

- can_lss_slave.c – LSS slave protocols.
- can_obdsdo_server.c – modules manager of the SDO server parameters object dictionary.
  can_obdsdo_server_default.h – SDO server default parameter.
  can_obdsdo_server_num.h – SDO server several parameters.
- can_obj_device.c – the device specification object dictionary.
- can_obdsrv.c – access manager to the server (slave) object dictionaries.
- can_server.c – modules manager for the complete SDO server transactions.
  can_server_block.h – SDO block protocol server transactions.
  can_server_common.h – common functions for the complete SDO server transactions.
  can_server_min.h – complete transactions for the SDO server default parameter.
  can_server_standard.h – complete transactions for the SDO server several parameters.

The following modules in this directory may be edited by the user:

- __can_devices.c – devices description modules manager.
  __can_device_*.h – slave devices description.
- __obd_mans_slave.c – application profile modules manager.
  __obdms_slave_*.h – application profile object dictionaries.

common directory. CANopen common modules.

\pdomapping sub-directory contains an object dictionary PDO mapping.

- can_backinit.c – CAN device (re)initialization functions, CANopen timer manager and monitor.
- can_canid.c – dynamic or static CAN–IDs processing manager.
  can_canid_dynamic.h – dynamic CAN–IDs and incoming CAN frames acceptance filter.
  can_canid_static.h – static CAN–IDs support.
- can_globals.c – external (global) variables and data structures.
- can_inout.c – CAN data link frames receiving and sending, the primary analysis of the received frames.
- can_led_indicator.c – LED indication.
- can_lib.c – general purpose functions: CRC calculation, data conversion, etc.
- can_malloc.c – specialized signal-safe dynamic memory allocation function.
- can_nmt_master.c – NMT master.
- can_nmt_slave.c – NMT slave.
- can_obj_deftype.c – type definition objects (DEFTYPE).
- can_obj_emcy.c – emergency producer objects (EMCY).
- can_obj_errors.c – error objects.
- can_obj_err_behaviour.c – error behavior object.

- can_obj_re_store.c – store/restore objects.
- can_obj_sync.c – SYNC objects.
- can_obj_time.c – time stamp object.
- can_pdo_map.c – dynamic or static PDO mapping modules manager.
  can_pdo_map_dynamic_bit.h – assembly, activation of the dynamic PDO bit-mapping.
  can_pdo_map_dynamic_byte.h – assembly, activation of the dynamic PDO byte-mapping.
  can_pdo_map_static.h – assembly, activation of the static PDO byte-mapping.
- can_pdo_obd.c – PDO communication parameters object dictionary.
- can_pdo_proc.c – processing the received and transmitted PDO frames.
- can_sdo_proc.c – processing the received and transmitted SDO frames.

The following modules in this directory may be edited by the user:

- __can_events.c – CANopen event handlers (emergency, errors, etc).
- __can_init.c – CAN node-ID and CAN bit rate index selection.
- pdomapping__map__static.h – static PDO mapping configuring.
- pdomapping__map_recv_*_*.h , pdomapping__map_tran_*_*.h – dynamic RPDO and TPDO mapping configuring.

include directory. Definitions and prototypes.

- can_header.h – main header module.
- can_genhead.h – basic includes and external definitions module.
- can_defines.h – CANopen specific constants and parameters.
- can_macros.h – CANopen library macros definitions.
- can_structures.h – data structures definitions.
- can_typedefs.h – data types definitions.
- can_defunc.h – common internal function prototypes.
- can_defunc_master.h – master internal functions prototypes.
- can_defunc_nmt.h – NMT functions prototypes.
- can_defunc_slave.h – slave internal functions prototypes.
- can_user_api_call.h – prototypes of the API functions, called by the user application.
- can_user_api_edit.h – prototypes of the API functions, called by the CANopen events.

The following modules in this directory may be edited by the user:

- __can_defines.h – CANopen configuration constants and parameters definitions.
- __can_defunc_master.h – master functions of the slave devices object dictionary mapping.
- __can_node_id.h – CAN node-ID and CAN bit rate index values.

root CANopen directory.

Runtime environment and OS dependent functions: CANopen timer, delays, critical sections etc.

- can_system_linux.h – system–dependent functions for Linux OS.
- can_system_windows.h – system–dependent functions for Windows OS.

The following modules in this directory may be edited by the user:

- _can_main.c – main(…) executable function and the library program main loop.
- __can_system.c – system–dependent modules manager.

The functionality of the library modules

The library units can belong to multiple functional groups.

CAN network data link layer modules

- can_canid.c – dynamic or static CAN–IDs processing manager.
  can_canid_dynamic.h – dynamic CAN–IDs and incoming CAN frames acceptance filter.
  can_canid_static.h – static CAN–IDs support.
- can_inout.c – CAN data link frames receiving and sending, the primary analysis of the received frames.
- can_test_driver.c – loopback test mode CAN driver.

SDO transaction modules

SDO protocol data exchange is initiated and managed by the SDO client. Therefore, the SDO client transactions are implemented by two-level scheme. Basic transaction is a confirmed service, when the client transmits request to the server and receives confirmation (both a single CAN frame). Complete SDO client transaction controls the entire cycle of data exchange.
The SDO server does not provide an explicit identification of basic and complete transactions. At the same time, the server monitors the entire data exchange in the SDO protocol.

- can_client.c – complete SDO client transactions support.
  can_clt_block.h – block protocol SDO client transactions.
- can_cltrans.c – basic SDO client transactions support (client request, server response).
- can_server.c – modules manager for the complete SDO server transactions.
  can_server_block.h – SDO block protocol server transactions.
  can_server_common.h – common functions for the complete SDO server transactions.
  can_server_min.h – complete transactions for the SDO server default parameter.
  can_server_standard.h – complete transactions for the SDO server several parameters.

CANopen objects assembly and processing modules

- can_pdo_proc.c – processing the received and transmitted PDO frames.
- can_sdo_proc.c – processing the received and transmitted SDO frames.
- can_obj_emcy.c – emergency producer (EMCY).
- can_nmt_master.c – NMT master objects reception and transmission.
- can_nmt_slave.c – NMT slave objects reception and transmission.
- can_pdo_map.c – dynamic or static PDO mapping modules manager.
  can_pdo_map_dynamic_bit.h – assembly, activation of the dynamic PDO bit-mapping.
  can_pdo_map_dynamic_byte.h – assembly, activation of the dynamic PDO byte-mapping.
  can_pdo_map_static.h – assembly, activation of the static PDO byte-mapping.

Communication profile object dictionary

- can_obdclt.c – access manager to the client (master) object dictionaries.
- can_obdsrv.c – access manager to the server (slave) object dictionaries.
- can_obj_device.c – the device specification object dictionary.
- __can_devices.c – devices description modules manager.
  __can_device_*.h – slave devices description (objects 1000_h, 1002_h, 1008_h, 1009_h, $100A_h$,

1018_h).
- can_obdsdo_client.c – client SDO parameters object dictionary (objects 1280_h to $12FF_h$).
- can_obdsdo_server.c – modules manager of the SDO server parameters object dictionary.
  can_obdsdo_server_default.h – SDO server default parameter (object 1200_h).
  can_obdsdo_server_num.h – SDO server several parameters (objects 1200_h to $127F_h$).
- can_nmt_master.c – NMT master objects reception and transmission (1016_h, $100C_h$, $100D_h$).
- can_nmt_slave.c – NMT slave objects reception and transmission (1017_h, $100C_h$, $100D_h$).
- can_obj_deftype.c – type definition objects (0001_h to 0007_h).
- can_obj_emcy.c – emergency producer objects (1014_h, 1015_h).
- can_obj_errors.c – error objects (1001_h, 1003_h).
- can_obj_err_behaviour.c – error behavior object (1029_h).
- can_obj_re_store.c – store/restore objects (1010_h, 1011_h).
- can_obj_sync.c – SYNC objects (1005_h, 1006_h, 1007_h, 1019_h).
- can_obj_time.c – time stamp object (1012_h).
- can_pdo_obd.c – PDO communication parameters object dictionary (objects 1400_h to $15FF_h$ and 1800_h to $19FF_h$).
- can_pdo_map.c – dynamic or static PDO mapping modules manager (objects 1600_h to $17FF_h$, and $1A00_h$ to $1BFF_h$).
  can_pdo_map_dynamic_bit.h – assembly, activation of the dynamic PDO bit-mapping.
  can_pdo_map_dynamic_byte.h – assembly, activation of the dynamic PDO byte-mapping.
  can_pdo_map_static.h – assembly, activation of the static PDO byte-mapping.
  \pdomapping__map__static.h – static PDO mapping objects configuring.
  \pdomapping__map_recv_*_*.h , pdomapping__map_tran_*_*.h – dynamic receive and transmit PDO mapping objects configuring.

Application profiles object dictionary

- __can_test_application.c – client operations and the test device object dictionary mapping.
- __obd_mans_master.c – access manager to the slave devices object dictionary mapping.
  __obdms_master_*.h – slave devices object dictionary mapping.
- __obd_mans_slave.c – application profiles access manager.
  __obdms_slave_*.h – application profile object dictionaries.

General purpose modules

- can_globals.c – external (global) variables and data structures.
- can_lib.c – general purpose functions: CRC calculation, data conversion, etc.
- can_malloc.c – specialized signal-safe dynamic memory allocation function.

Initialization and events processing modules

- can_backinit.c – CAN device (re)initialization functions, CANopen timer manager and monitor.
- __can_events.c – CANopen event handlers (emergency, errors, etc).
- __can_init.c – CAN node-ID and CAN bit rate index selection.

Other modules

- can_led_indicator.c – LED indication.
- can_lss_slave.c – LSS slave protocols.
- __can_main.c – main(…) executable function and the library program main loop.

- __can_system.c – system–dependent modules manager.
  can_system_linux.h – system–dependent functions for Linux OS.
  can_system_windows.h – system–dependent functions for Windows OS.

CAN data link layer driver API

CANopen library connects to the data link layer using CHAI driver. Only the basic driver API functions are involved, that present in all its versions. This section explains the driver functions purpose to facilitate its development for other target platforms.

_s16 CiInit(void);
Performs the initialization of the CAN controller hardware. Initializes the driver data structures. The function executes once when starting CAN device.
Called from can_backinit.c module.
Return values: normal completion = 0; error < 0.

_s16 CiOpen(_u8 chan, _u8 flags);
Initializes CAN controller channel **chan** in non-blocking mode with 11-bit CAN-IDs processing. The function sets hardware modes of the CAN controller channel. Initializes data structures for this channel, as well as hardware or software acceptance filter.
Called from can_backinit.c module.
Parameters:
- **chan** – CAN controller channel number (starts from 0).
- **flags** – specifies the type of processed CAN identifiers (11-bit and/or 29-bit).

Return values: normal completion = 0; error < 0.

_s16 CiClose(_u8 chan);
Closes CAN channel **chan**. Disables the interrupts, resets the registers, removes the signal handlers. The sequence of function calls CiClose(...) → CiOpen(...) performs reinitialization of the CAN controller channel.
Called from can_backinit.c module.
Parameters:
- **chan** – CAN controller channel number (starts from 0).

Return values: normal completion = 0; error < 0.

_s16 CiStart(_u8 chan);
Changes the controller channel **chan** to an active state, enabling hardware interrupts.
Called from can_backinit.c and can_canid_dynamic.h modules.
Parameters:
- **chan** – CAN controller channel number (starts from 0).

Return values: normal completion = 0; error < 0.

_s16 CiStop(_u8 chan);
Changes the controller channel **chan** to inactive state, disabling hardware interrupts.
Called from can_backinit.c and can_canid_dynamic.h modules.
Parameters:
- **chan** – CAN controller channel number (starts from 0).

Return values: normal completion = 0; error < 0.

_s16 CiSetFilter(_u8 chan, _u32 acode, _u32 amask);
_s16 CiSetDualFilter(_u8 chan, _u32 acode0, _u32 amask0, _u32 acode1, _u32 amask1);
Sets one or two-level bit-mask acceptance filter. Mask filter can be implemented in hardware, if supported by CAN controller. However, the performance of modern microcontrollers is sufficient to implement the acceptance filter in the software driver.
Called from can_canid_dynamic.h module.

Parameters:
- **chan** – CAN controller channel number (starts from 0).
- **acode0** – the required bit values for the first filter.
- **mask0** – first filter bit mask (1 = the value of the corresponding bit **acode0** taken into account, 0 = the value is ignored).
- **acode1** – the required bit values for the second filter.
- **amask1** – second filter bit mask (1 = the value of the corresponding bit **acode1** taken into account, 0 = the value is ignored).

Return values: normal completion = 0; error < 0.

_s16 CiSetBaud(_u8 chan, _u8 bt0, _u8 bt1);
Sets CAN network bit rate for the controller channel **chan**.
Called from can_backinit.c module.
Parameters:
- **chan** – CAN controller channel number (starts from 0).
- **bt0, bt1** – CAN network bit rate codes which values depend on the CAN controller type.

Return values: normal completion = 0; error < 0.

_s16 CiWrite(_u8 chan, canmsg_t *mbuf, _s16 cnt);
Writes to the controller **chan** buffer (or output driver queue) one CAN data link layer frame. The writing if done in non-blocking mode. To improve the dynamic performance of the library it is recommended to set zero write timeout. For application functions the CANopen library provides signal-safe, re-entrant CAN frames writing into the program cache.
Called from can_inout.c module.
Parameters:
- **chan** – CAN controller channel number (starts from 0).
- ***mbuf** – a pointer to the CAN data link layer frame structure.
- **cnt** – the number of frames to write. For the CANopen library is always 1.

Return values: normal completion = 1 (the number of actually written CAN frames); error <= 0.

_s16 CiRead(_u8 chan, canmsg_t *mbuf, _s16 cnt);
Reads from the controller **chan** buffer (or input driver queue) one CAN data link layer frame to be processed by the CANopen library. Called from a CAN frame receiving event handler.
Called from can_inout.c module.
Parameters:
- **chan** – CAN controller channel number (starts from 0).
- ***mbuf** – a pointer to the CAN data link layer frame structure.
- **cnt** – the number of frames to read. For the CANopen library is always 1.

Return values: normal completion = 1 (the number of actually read CAN frames); error <= 0.

_s16 CiSetCB(_u8 chan, _u8 ev, void (*ci_handler) (_s16));
Registers signal (event) handler to receive CAN frames for channel **chan**. The handler is signal-safe and can be called directly from the controller hardware interrupts. The handler provides sequential read of the frames, taken in the controller buffer (or input driver queue) during the processing of the current CAN frame (re-entrant mode). Note that each handler call performs significant volume of the software code.
Called from can_backinit.c module.
Parameters:
- **chan** – CAN controller channel number (starts from 0).
- **ev** – the event for which the handler is set (reception of a CAN frame).
- ***ci_handler** – pointer to the can_read_handler(…) function, that handlers the received frames.

The function is placed in the can_inout.c module.
Return values: normal completion = 0; error < 0.

_s16 CiSetCB(_u8 chan, _u8 ev, void (*ci_handler) (_s16));
Registers error events handler for channel **chan**. The handler is signal-safe and can be called directly from the controller hardware interrupts. In case of overlapping requests to the handler (re-entrant mode) loss of records in the list of pre-defined errors (object 1003_h) is possible. But anyway, the information will be stored in the error register (object 1001_h) Note that each handler call performs significant volume of the software code.
Called from can_backinit.c module.
Parameters:
- **chan** – CAN controller channel number (starts from 0).
- **ev** – the event for which the handler is set (error signal).
- ***ci_handler** – pointer to the consume_controller_error(…) function, that handlers the error.
The function is placed in the __can_event.c. module.

Return values: normal completion = 0; error < 0.

void ci_propogate_sigs(void);
Propagation function of the CAN driver signals.
If the driver does not provide asynchronous delivery of any incoming signals (events) to the library, the propagator must be included in the main program loop. Herewith, the events processing latency depends on the total duration of the main loop.

Communication profile area

All communication objects supported by the CANopen library are presented. Library modules placement is given relative to the "root" CANopen directory.

NMT objects

- **$100C_h$**
  Guard time, milliseconds.
  Module \common\can_nmt_master.c for the NMT master and \common\can_nmt_slave.c for the NMT slave. NMT master object is represented by an array.
- **$100D_h$**
  Life time factor.
  Module \common\can_nmt_master.c for the NMT master and \common\can_nmt_slave.c for the NMT slave. NMT master object is represented by an array.
- **1016_h**
  Consumer heartbeat time.
  The heartbeat time is given in multiples of milliseconds. Module \common\can_nmt_master.c. The number of this object sub-indexes is determined by the parameter CAN_NOF_NODES (the number of CAN nodes). The heartbeat protocol is of higher priority than the node guarding protocol. The consumer heartbeat time is initialized for each node by the default value CAN_HBT_CONSUMER_MS. To enable the node guarding protocol, both consumer and producer heartbeat time must be set to zero.
- **1017_h**
  Producer heartbeat time.
  The heartbeat time is given in multiples of milliseconds. Module \common\can_nmt_slave.c. The heartbeat protocol is of higher priority than the node guarding protocol. The producer heartbeat time is initialized by the default value CAN_HBT_PRODUCER_MS. To enable the node guarding protocol, both consumer and producer heartbeat time must be set to zero.

Master and slave objects

- **1005_h**
  COB–ID SYNC message.
  Module \common\can_obj_sync.c
- **1006_h**
  Communication cycle period.
  The value is given in multiples of microseconds. Module \common\can_obj_sync.c
- **1007_h**
  Synchronous window length.
  The time window for synchronous PDOs. The value is given in multiples of microseconds. Module \common\can_obj_sync.c.
- **1012_h**
  COB–ID time stamp object.
  Module \common\can_obj_time.c
- **1019_h**
  Synchronous counter overflow value.
  Module \common\can_obj_sync.c.

- **1029_h**
  Error behavior object.
  Supported for the NMT slave. Module \common\can_obj_err_behaviour.c.
- **1400_h** to **$15FF_h$**
  RPDO communication parameter.
  The module \common\can_pdo_obd.c contains up to 512 RPDO communication parameters. Actual number of the receive PDOs is defined by the parameters CAN_NOF_PDO_RECV_SLAVE for the slave and CAN_NOF_PDO_RECV_MASTER for the master. RPDOs are processed in the \common\can_pdo_proc.c module.
- **1600_h** to **$17FF_h$**
  RPDO mapping parameter.
  For the dynamic PDO bit-mapping the module \common\can_pdo_map_dynamic_bit.h contains up to 512 RPDO mapping parameters. As required the module loads mapping definitions from the \pdomapping sub-directory. The definition manager is in the module \pdomapping\can_mappdo_main.h and the mapping definitions are packaged by 32 RPDOs in each \pdomapping__map_recv_*_*.h file. Dynamic PDO byte-mapping object dictionary is created in the \common\can_pdo_map_dynamic_byte.h module. Static PDO byte-mapping is defined in the modules \common\can_pdo_map_static.h and \pdomapping__map__static.h.
- **1800_h** to **$19FF_h$**
  TPDO communication parameter.
  The module \common\can_pdo_obd.c contains up to 512 TPDO communication parameters. Actual number of the transmit PDOs is defined by the parameters CAN_NOF_PDO_TRAN_SLAVE for the slave and CAN_NOF_PDO_TRAN_MASTER for the master. TPDOs are processed in the \common\can_pdo_proc.c module.
- **$1A00_h$** to **$1BFF_h$**
  TPDO mapping parameter.
  For the dynamic PDO bit-mapping the module \common\can_pdo_map_dynamic_bit.h contains up to 512 TPDO mapping parameters. As required the module loads mapping definitions from the \pdomapping sub-directory. The definition manager is in the module \pdomapping\can_mappdo_main.h and the mapping definitions are packaged by 32 TPDOs in each \pdomapping__map_tran_*_*.h file. Dynamic PDO byte-mapping object dictionary is created in the \common\can_pdo_map_dynamic_byte.h module. Static PDO byte-mapping is defined in modules \common\can_pdo_map_static.h and \pdomapping__map__static.h.

Master objects

- **1280_h** to **$12FF_h$**
  SDO client parameter.
  The module \master\can_obdsdo_client.c contains up to 128 SDO client parameters. Actual number of the client SDOs is defined by the parameter CAN_NOF_NODES (the number of CAN nodes). SDOs are processed in the \common\can_sdo_proc.c module.

Slave objects

- **1000_h**
  Device type.
  Modules \slave\can_obj_device.c, \slave__can_devices.c, \slave__can_device_*.h.
- **1001_h**
  Error register.
  Module \common\can_obj_errors.c.

- **1002_h**
  Manufacturer status register.
  Modules \slave\can_obj_device.c, \slave__can_devices.c, \slave__can_device_*.h.
- **1003_h**
  Pre–defined error field.
  Module \common\can_obj_errors.c.
- **1008_h**
  Manufacturer device name.
  Modules \slave\can_obj_device.c, \slave__can_devices.c, \slave__can_device_*.h.
- **1009_h**
  Manufacturer hardware version.
  Modules \slave\can_obj_device.c, \slave__can_devices.c, \slave__can_device_*.h.
- **$100A_h$**
  Manufacturer software version.
  Modules \slave\can_obj_device.c, \slave__can_devices.c, \slave__can_device_*.h.
- **1010_h**
  Store parameters.
  Module \common\can_obj_re_store.c.
- **1011_h**
  Restore default parameters.
  Module \common\can_obj_re_store.c.
- **1014_h**
  COB–ID EMCY.
  Module \common\can_obj_emcy.c.
- **1015_h**
  Inhibit time EMCY.
  The value is given in multiples of 100 microseconds. Module \common\can_obj_emcy.c.
- **1018_h**
  Identity object.
  Modules \slave\can_obj_device.c, \slave__can_devices.c, \slave__can_device_*.h.
- **1200_h** to **$127F_h$**
  SDO server parameter.
  To optimize target application, the library includes two versions of the SDO server object dictionary modules. In case of utilizing the only SDO server default parameter the module \slave\can_obdsdo_server_default.h is used. If the server supports 2 to 128 SDO parameters the module \slave\can_obdsdo_server_num.h is applied. Access manager to the SDO server parameter object dictionary is the \slave\can_obdsdo_server.c module. Actual number of the server SDOs is defined by the parameter CAN_NOF_SDO_SERVER. SDO server parameters are created in the object dictionary starting with index 1200_h. SDOs are processed in the \common\can_sdo_proc.c module.

CANopen application assembly settings

The parameters are defined in modules \include__can_defines.h and \include__can_node_id.h.
Important note.
In most cases, library units do not control values and ranges of the application assembly parameters and settings. Therefore, any change should be made only with the case knowledge and the ability to cope with the consequences.

- CAN_APPLICATION_MODE
  The target application general build mode:
  MASTER – building the application for the master device (SDO client).
  SLAVE – building for the slave device (SDO server).
  TEST – test mode. Used for debugging the library in loopback mode. Does not require a CAN controller and data link layer driver.
- CAN_NMT_MODE
  The target application network management (NMT) build mode:
  MASTER – NMT master device.
  SLAVE – NMT slave device.
- CAN_SLAVE_DEVICE_CLASS
  Defines the slave application device profile. It is expected that the software modules supporting appropriate profile are available.
- CAN_NETWORK_CONTROLLER
  CAN controller channel number. The default value.
- CAN_BITRATE_INDEX
  CAN bit rate index. The default value.
- CAN_OS_LINUX, CAN_OS_WIN32
  The type of operating system for which the CANopen application is built.
  CAN_OS_LINUX – Linux.
  CAN_OS_WIN32 – Microsoft Windows.
- CAN_ID_MODE
  CAN data link identifier size.
  CANID11 – 11-bit CAN–ID.
  CANID29 – 29-bit CAN–ID (reserved, not used in CANopen).
- CAN_FRAME_READ_MODE
  Defines the method of receiving CAN data link frame from the driver.
  SIGNAL – frames are read by the signal (for the general purpose operating systems) or CAN controller hardware interrupt.
  POLL – CAN frames are read by polling from the program main loop.
- CAN_BYTE_ORDER
  Byte order for numeric data types (integer, real). The data is transferred across the network starting with the least significant byte.
  NORMAL – little-endian.
  REVERSE – big-endian.
- CAN_PDO_MAPPING_MODE
  Sets PDO mapping method.
  DYNAMIC – dynamic PDO mapping.
  STATIC – static, non-modifiable PDO mapping.
  Dynamic PDO mapping is either bit or byte intended. Static PDO mapping is byte-specific, that

is, one PDO can contain up to eight objects, the size of each is a multiple of 8 bits. The maximum number of application objects that are mapped dynamically in one PDO can be defined individually for each RPDO and TPDO in the range from 1 to 64, taking into account the granularity (CiA 306). The default value is specified by the CAN_NOF_MAP parameter. For the static, non-modifiable PDO mapping granularity is zero.

- CAN_DYNAMIC_MAPPING_GRANULARITY
  Granularity flag of the dynamic PDO mapping.
  MAPBIT – up to 64 application objects can be mapped into one PDO.
  MAPBYTE – one PDO can contain up to eight objects, the size of each is a multiple of 8 bits.
- CAN_MASTER_RECVCANID_METHOD
  The CAN–IDs processing method for the CANopen master.
  DYNAMIC – the dynamic method (linear search).
  STATIC – the static method (direct addressing).
- CAN_HARD_ACCEPTANCE_FILTER
  CAN controller bit-mask acceptance filter.
  AFSINGLE – single-level filtering.
  AFDUAL – double-level filtering.
- CAN_LED_INDICATOR
  LED indication mode (CiA 303 p. 3)
  COMBINED – one bicolor red/green LED is used.
  SEPARATE – two single LEDs (red and green) are used.
- CAN_CRC_MODE
  CRC calculation method.
  CRCTABLE – byte-optimized CRC calculation.
  CRCDIRECT – bit-polynomial CRC calculation.
- CAN_OBJECT_EMCY
  Emergency object support.
  TRUE – EMCY object exists.
  FALSE – the object does not exist.
- CAN_OBJECT_TIME
  Time stamp object support.
  TRUE – time stamp object exists.
  FALSE – the object does not exist.
- CAN_OBJECT_RE_STORE
  Non-volatile memory storage objects support.
  TRUE – store/restore objects exist.
  FALSE – the objects do not exist.
  The object is also maintained when LSS protocol activated.
- CAN_OBJECT_ERR_BEHAVIOUR
  Error behavior object support.
  TRUE – the object is supported.
  FALSE – the object is not supported.
- CAN_PROTOCOL_BLOCK
  SDO block protocol.
  TRUE – SDO block protocol supported.
  FALSE – block protocol is not supported.
- CAN_PROTOCOL_LSS
  LSS protocol.
  TRUE – LSS protocol supported.

FALSE – LSS protocol is not supported.
When LSS protocol is active, non-volatile memory storage object is also activated.

- CAN_NOF_NODES
  The number of nodes in the CAN network.
  Master objects CAN-IDs are initialized with the generic pre-defined connection set, assuming that the CAN nodes are numbered sequentially. For example, CAN network with three nodes is configured for node-IDs 1, 2 and 3. The user application can change this configuration.
- CAN_NOF_RECVCANID_SLAVE
  The total number of CAN-IDs, supported by the slave. It uses only the dynamic processing method.
- CAN_NOF_RECVCANID_MASTER
  The total number of CAN-IDs, supported by the master with the dynamic processing method.
- CAN_NOF_PREDEF_ERRORS
  The maximum number of errors registered for the pre–defined error field (object 1003_h).
- CAN_NOF_ERRBEH_SUBIND
  Error behavior object highest sub-index.
- CAN_NOF_MAP
  The maximum number of application objects that can be mapped dynamically in one PDO taking into account the granularity: 1 to 64. For each receive and transmit PDO the parameter can be set individually in the mapping configuring files: \common\pdomapping__map_recv_*_*.h for RPDOs and \common\pdomapping__map_tran_*_*.h for TPDOs.
- CAN_NOF_SDO_SERVER
  The number of SDO server parameters.
  Server SDOs are initialized with the generic pre-defined connection set, taking into account CAN node-ID. The user application can change this configuration.
- CAN_NOF_PDO_RECV_SLAVE
  CAN_NOF_PDO_TRAN_SLAVE
  The number of the slave RPDOs.
  The number of the slave TPDOs.
  Slave PDOs are initialized with the generic pre-defined connection set, taking into account CAN node-ID. The user application can change this configuration.
- CAN_NOF_SYNCPDO_MASTER
  The size of each FIFO buffer for the master synchronous RPDOs and TPDOs.
- CAN_TIMERUSEC
  The CANopen timer period in microseconds.
  The parameter value should be not less than 100 ms. The timer period can be changed depending on the resolution requirements for various CANopen objects: SYNC, PDO event timer, inhibit times, etc.
- CAN_TIMEOUT_RETRIEVE
  CAN network data retrieve timeout for the basic SDO client transaction. Specified in microseconds. In a basic transaction the client transmits request to the server and receives confirmation.
- CAN_TIMEOUT_READ
  Application data read timeout for the basic SDO client transaction. Specified in microseconds.
- CAN_TIMEOUT_SERVER
  SDO server basic transaction timeout. Specified in microseconds. In a basic transaction the server is waiting data request from the client.
- CAN_HBT_PRODUCER_MS
  The default value of the producer heartbeat time. Initializes object 1017_h.

- CAN_HBT_CONSUMER_MS
  The default value of the consumer heartbeat time. Initializes object 1016_h for each node.
- CAN_EMCY_INHIBIT_100MCS
  The default value of the inhibit time EMCY. Initializes object 1015_h.
- CAN_RPDO_TRTYPE
  The default value of the RPDO transmission type. Initializes sub-index 2 of the objects 1400_h to $15FF_h$ – RPDO communication parameter.
- CAN_TPDO_TRTYPE
  The default value of the TPDO transmission type. Initializes sub-index 2 of the objects 1800_h to $19FF_h$ – TPDO communication parameter.
- CAN_TPDO_INHIBIT_100MCS
  The default value of the TPDO inhibit time. Initializes sub-index 3 of the objects 1800_h to $19FF_h$ – TPDO communication parameter.
- CAN_RPDO_ET_MS
  The default value of the RPDO event timer. Initializes sub-index 5 of the objects 1400_h to $15FF_h$ – RPDO communication parameter.
- CAN_TPDO_ET_MS
  The default value of the TPDO event timer. Initializes sub-index 5 of the objects 1800_h to $19FF_h$ – TPDO communication parameter.
- CAN_TPDO_SYNC_START
  The default PDO SYNC start value. Initializes sub-index 6 of the objects 1800_h to $19FF_h$ – TPDO communication parameter.

- CAN_SIZE_MAXSDOMEM
  The maximum size of the object dictionary entry, transferred consistently using the segmented SDO protocol. This parameter is used by the specialized signal-safe dynamic memory allocation function to determine the maximum buffer size. If the length of the object dictionary entry exceeds the specified size, it can only be transferred using a non-buffer mode of the SDO block protocol. As a rule, block SDO transfer is performed directly between the object dictionary entries of the transmitting and receiving parties. This is provided by an access to the relevant entries using a byte pointer. Block protocol ensures the received data consistency only after the successful completion of the entire data exchange cycle. Otherwise, the corresponding object dictionary entry should not be used.
- CAN_LEN_VISIBLE_STRING
  The maximum length of the visible string data.

- CAN_NODEID_SLAVE
  CANopen slave node-ID default number. Valid values are 1 to 127 and 255.
- CAN_SERIAL_NUMBER
  CANopen slave device serial number (object $1018_h sub4_h$).

Master and slave functions API

int16 pdo_remote_transmit_request(canindex index);
Generates and sends remote transmit request (RTR) for the PDO, defined by the communication parameter **index**. PDO must be defined as RPDO (receive), be valid and RTR allowed on this PDO. All application objects that are mapped in the corresponding RPDO, should be available at read and write access.
Parameters:
- **index** – RPDO communication parameter index.

Return values: normal completion = 0; error < 0.
- CAN_RETOK – normal completion.
- CAN_ERRET_COMM_SEND – failed to send frame in the CAN network.
- CAN_ERRET_NODE_STATE – the CAN node is not in operational state.
- CAN_ERRET_OBD_NOOBJECT – PDO configuring error: CANopen object with **index** does not exist or it is not RPDO.
- CAN_ERRET_PDO_INVALID – PDO is not valid.
- CAN_ERRET_PDO_NORTR – no RTR allowed on this PDO.

int16 transmit_can_pdo(canindex index);
Generates and sends TPDO, defined by the communication parameter **index** and with the following transmission types:
- 0 – synchronous (acyclic);
- 254, 255 – event-driven;

PDO must be defined as TPDO (transmit), be valid and not inhibited.
Parameters:
- **index** – TPDO communication parameter index.

Return values: normal completion = 0; error < 0.
- CAN_RETOK – normal completion.
- CAN_ERRET_COMM_SEND – failed to send frame in the CAN network.
- CAN_ERRET_NODE_STATE – the CAN node is not in operational state.
- CAN_ERRET_OBD_NOOBJECT – PDO configuring error: CANopen object with **index** does not exist or it is not TPDO.
- CAN_ERRET_PDO_INVALID – PDO is not valid.
- CAN_ERRET_PDO_ERRMAP – incorrect object size is specified in the PDO mapping or the total length of the mapped objects exceeds the maximum PDO size (64 bits).
- CAN_ERRET_PDO_INHIBIT – PDO is inhibited.
- CAN_ERRET_PDO_MAP_DEACT – PDO mapping is deactivated.
- CAN_ERRET_PDO_TRTYPE – invalid PDO transmission type.

void produce_time(unsigned32 ms, unsigned16 days);
Generates and sends time stamp object. It is formed in accordance with the TIME_OF_DAY data type.
Parameters:
- **ms** – the time in milliseconds after midnight (28 LSB used).
- **days** – the number of days since January 1, 1984.

Master functions API

void can_sdo_client_transfer(struct sdocltappl *ca);
Performs complete SDO client transaction. Its modes, conditions and results are contained in the ***ca** structure.
Parameters:

- **ca.operation** – defines the basic SDO transfer mode. Specified by the user and modified by the function. CAN_SDOPER_DOWNLOAD – download or CAN_SDOPER_UPLOAD – upload. If the data size does not exceed 4 bytes, expedited SDO protocol is used. When the data size is more than 4 bytes but not exceeding CAN_SIZE_MAXSDOMEM, segmented SDO transfer is utilized. With a larger amount of data SDO block protocol is used. After the function execution, the parameter **ca.operation** contains the mode actually used when SDO transfer:
  CAN_SDOPER_(UP/DOWN)_EXPEDITED – expedited,
  CAN_SDOPER_(UP/DOWN)_SEGMENTED – segmented,
  CAN_SDOPER_(UP/DOWN)_BLOCK – block mode.
  The basic mode itself (UPload or DOWNload) is always the same.
- **ca.node** – SDO server node-ID. Determined independently by the function and is not set by the user. Contains the slave device node-ID to access the application profile objects, mapped in the master. Extracted by the function from the SDO client communication parameter (objects 1280_h to $12FF_h$).
- **ca.datasize** – the size of transferred data in bytes. Set by the user or determined by the function. Must be specified when the pointer **ca.datapnt** is not NULL and in this case not changed by the function. When the value of the **ca.datapnt** pointer is NULL, **ca.datasize** is determined by the function independently.
- **ca.datapnt** – local buffer pointer. Can be set by the user. If the pointer is not set (NULL) and SDO block protocol is used, it is determined by the function; otherwise it is not changed. If the pointer **ca.datapnt** is set (not NULL), the transferred SDO data will be read from or written to the local buffer, defined by the pointer. In this case **ca.datasize** should be specified. When the pointer **ca.datapnt** is not set (NULL) slave application object, mapped in the master is used. The object entry is determined by the parameters **ca.node** (SDO server / slave node-ID), **ca.si.index** (application object index) and **ca.si.subind** (application object sub-index).
- **sdoixs** – SDO indexes structure. Specified by the user and is not modified by the function. **ca.si.sdoind** – SDO client communication parameter index (1280_h to $12FF_h$). **ca.si.index** and **ca.si.subind** accordingly the index and sub-index of the application object, transferred with the SDO.
- **sdostatus** – SDO client transaction status. Set only by the function and contains the transaction exit code. **ca.ss.state** – the client transaction exit status. **ca.ss.abortcode** – SDO abort code, if the **ca.ss.state** has the value CAN_TRANSTATE_SDO_SRVABORT. If the transaction is successful, the exit status is CAN_TRANSTATE_OK. Otherwise, one of the error codes is set:
  CAN_TRANSTATE_OBD_ZERO – the object dictionary entry has zero size;
  CAN_TRANSTATE_OBD_READ – object dictionary reading error;
  CAN_TRANSTATE_OBD_WRITE – object dictionary writing error;
  CAN_TRANSTATE_OBD_NOOBJECT – object does not exist in the object dictionary;
  CAN_TRANSTATE_OBD_NOSUBIND – sub-index does not exist;
  CAN_TRANSTATE_OBD_MALLOC – dynamic buffer allocating error;
  CAN_TRANSTATE_SDO_RETRANSMIT – the number of data segment retransmissions exceeded (block protocol);
  CAN_TRANSTATE_SDO_BLKSIZE – incorrect number of segments in the data block (block protocol);
  CAN_TRANSTATE_SDO_SEQNO – incorrect segment number (block protocol);
  CAN_TRANSTATE_SDO_CRC – CRC error (block protocol);

CAN_TRANSTATE_SDO_SUB – incorrect sub-command (block protocol);
CAN_TRANSTATE_SDO_TOGGLE – toggle bit error (segmented transfer);
CAN_TRANSTATE_SDO_DATASIZE – invalid data segment size;
CAN_TRANSTATE_SDO_OBJSIZE – the object size known to the client and the server does not match;
CAN_TRANSTATE_SDO_MODE – client and server transfer modes mismatch (SDO upload protocol);
CAN_TRANSTATE_SDO_MPX – client and server multiplexors mismatch;
CAN_TRANSTATE_SDO_SRVABORT – SDO abort received;
CAN_TRANSTATE_SDO_INVALID – SDO is invalid;
CAN_TRANSTATE_SDO_WRITERR – SDO transfer error in the CAN network;
CAN_TRANSTATE_SDO_SCSERR – SDO client received from the server invalid or unknown command;
CAN_TRANSTATE_SDO_TRANS_TIMEOUT – SDO client basic transaction internal timeout;
CAN_TRANSTATE_SDO_NET_TIMEOUT – SDO client basic transaction network timeout;
CAN_TRANSTATE_SDO_READ_TIMEOUT – data read in the application timeout, basic transaction was reset;
CAN_TRANSTATE_SDO_NOWORKB – SDO client basic transaction working buffer overflow;
CAN_TRANSTATE_SDO_NODE – error reading node-ID of the SDO server;
CAN_TRANSTATE_ERROR – generic error.

int16 get_pdo_node(canindex index, cannode *node);
Reading the node-ID for the PDO.
Parameters:
- ***node** – the node-ID for the PDO communication parameter **index**.
- **index** – PDO communication parameter index.

Return values: normal completion = 0; error < 0.
- CAN_RETOK – normal completion.
- CAN_ERRET_OBD_NOOBJECT – PDO object does not exist.

int16 put_pdo_node(canindex index, cannode node);
Writing the node-ID for the PDO.
Parameters:
- **node** – the node-ID for the PDO communication parameter **index**.
- **index** – PDO communication parameter index.

Return values: normal completion = 0; error < 0.
- CAN_RETOK – normal completion.
- CAN_ERRET_OBD_NOOBJECT – PDO object does not exist.

void nmt_master_command(unsigned8 cs, cannode node);
Generates and sends to the CAN network NMT frame that contains the NMT command **cs** and the CAN node number **node**. The function does not perform any checks of the NMT command and the node-ID values.
Parameters:
- **cs** – NMT command specifier.
- **node** – the node-ID.

canbyte *node_get_manstan_objpointer(cannode node, canindex index, cansubind subind);
Getting byte pointer to the slave application profile object, mapped in the master.
Parameters:
- **node** – the slave node-ID.
- **index** – mapped application object index.
- **subind** – mapped application object sub-index.

Return values:
- not NULL – byte pointer to the object defined by the function arguments.
- NULL – the object is not accessible through a pointer.

int16 node_see_manstan_access(cannode node, canindex index, cansubind subind);
Getting access bit mask to the slave application profile object, mapped in the master.
Parameters:
- **node** – the slave node-ID.
- **index** – mapped application object index.
- **subind** – mapped application object sub-index.

Return values: access bit mask >= 0; error < 0.
- CAN_MASK_ACCESS_PDO – PDO mapping allowed for this object (LSB = 1).
- CAN_MASK_ACCESS_RO – read access for the object (LSB+1 = 1).
- CAN_MASK_ACCESS_WO – write access for the object (LSB+2 = 1).
- CAN_MASK_ACCESS_RW – read and write access (LSB+1 = 1 and LSB+2 = 1).
- CAN_ERRET_OBD_INVNODE – invalid CAN node.
- CAN_ERRET_OBD_NOOBJECT – object does not exist in the object dictionary.
- CAN_ERRET_OBD_NOSUBIND – sub-index does not exist.

int32 node_get_manstan_objsize(cannode node, canindex index, cansubind subind, int16 unit);
Object size request from the slave application profile, mapped in the master. This function also detects the presence of the corresponding object in the dictionary. The size may be represented in bytes (**unit** = BYTES) or in bits (**unit** = BITS). The size in bits is used for bit-specific PDO mapping.
Parameters:
- **node** – the slave node-ID.
- **index** – mapped application object index.
- **subind** – mapped application object sub-index.
- **unit** – object size measurement unit: bytes (BYTES) or bits (BITS).

Return values: object size > 0; error < 0.
- > 0 – the size of the object in **unit**s.
- CAN_ERRET_OBD_INVNODE – invalid CAN node.
- CAN_ERRET_OBD_NOOBJECT – object does not exist in the object dictionary.
- CAN_ERRET_OBD_NOSUBIND – sub-index does not exist.

int16 node_get_manstan_objtype(cannode node, canindex index, cansubind subind);
Object type request from the slave application profile, mapped in the master.
Parameters:
- **node** – the slave node-ID.
- **index** – mapped application object index.
- **subind** – mapped application object sub-index.

Return values: object type > 0; error < 0.
- > 0 – static data type index (0001_h to $001F_h$).
- CAN_ERRET_OBD_INVNODE – invalid CAN node.

- CAN_ERRET_OBD_NOOBJECT – object does not exist in the object dictionary.
- CAN_ERRET_OBD_NOSUBIND – sub-index does not exist.

int16 node_read_manstan_objdict(cannode node, canindex index, cansubind subind, canbyte *data);
Reading an object from the slave application profile, mapped in the master. The result is converted into a byte format and located at ***data**. The byte order is not changed. The application must allocate a buffer large enough to hold the entire object. The size of the object can be determined using the function node_get_manstan_objsize(...).
Parameters:
- **node** – the slave node-ID.
- **index** – mapped application object index.
- **subind** – mapped application object sub-index.
- ***data** – byte pointer to the data.

Return values: normal completion = 0; error < 0.
- CAN_RETOK – normal completion.
- CAN_ERRET_OBD_INVNODE – invalid CAN node.
- CAN_ERRET_OBD_NOOBJECT – object does not exist in the object dictionary.
- CAN_ERRET_OBD_NOSUBIND – sub-index does not exist.
- CAN_ERRET_OBD_WRITEONLY – attempt to read a write only object.

int16 node_read_manstan_objdict_network(cannode node, canindex index, cansubind subind, canbyte *data);
Reading an object from the slave application profile, mapped in the master. The byte order is arranged in a network transfer sequence. If CAN_BYTE_ORDER = NORMAL the function fully corresponds to node_read_manstan_objdict(...). When CAN_BYTE_ORDER = REVERSE the byte order for the basic numeric data types is reversed.
Parameters:
see node_read_manstan_objdict(...).
Return values: normal completion = 0; error < 0.
see node_read_manstan_objdict(...).

int16 node_write_manstan_objdict(cannode node, canindex index, cansubind subind, canbyte *data);
Writing an object to the slave application profile, mapped in the master. The function places the object located at ***data** in the object dictionary entry. The byte order is not changed. Before calling the function, the application must convert the data into a byte format.
Parameters:
- **node** – the slave node-ID.
- **index** – mapped application object index.
- **subind** – mapped application object sub-index.
- ***data** – byte pointer to the data.

Return values: normal completion = 0; error < 0.
- CAN_RETOK – normal completion.
- CAN_ERRET_OBD_INVNODE – invalid CAN node.
- CAN_ERRET_OBD_NOOBJECT – object does not exist in the object dictionary.
- CAN_ERRET_OBD_NOSUBIND – sub-index does not exist.
- CAN_ERRET_OBD_READONLY – attempt to write a read only object.

int16 node_write_manstan_objdict_network(cannode node, canindex index, cansubind subind, canbyte *data);
Writing an object to the slave application profile, mapped in the master. The byte order is arranged in a local application sequence. If CAN_BYTE_ORDER = NORMAL the function fully corresponds to node_write_manstan_objdict(...). When CAN_BYTE_ORDER = REVERSE the byte order for the basic numeric data types is reversed.
Parameters:
see node_write_manstan_objdict(...).
Return values: normal completion = 0; error < 0.
see node_write_manstan_objdict(...).

int16 client_see_access(canindex index, cansubind subind);
Getting access bit mask to the client (master) communication profile object.
Parameters:
- **index** – communication object index.
- **subind** – communication object sub-index.

Return values: access bit mask >= 0; error < 0.
- CAN_MASK_ACCESS_PDO – PDO mapping allowed for this object (LSB = 1).
- CAN_MASK_ACCESS_RO – read access for the object (LSB+1 = 1).
- CAN_MASK_ACCESS_WO – write access for the object (LSB+2 = 1).
- CAN_MASK_ACCESS_RW – read and write access (LSB+1 = 1 and LSB+2 = 1).
- CAN_ERRET_OBD_NOOBJECT – object does not exist in the object dictionary.
- CAN_ERRET_OBD_NOSUBIND – sub-index does not exist.

int32 client_get_object_size(canindex index, cansubind subind, int16 unit);
Object size request from the client (master) communication profile. This function also detects the presence of the corresponding object in the dictionary. The size may be represented in bytes (**unit** = BYTES) or in bits (**unit** = BITS). The size in bits is used for bit-specific PDO mapping.
Parameters:
- **index** – communication object index.
- **subind** – communication object sub-index.
- **unit** – object size measurement unit: bytes (BYTES) or bits (BITS).

Return values: object size > 0; error < 0.
- > 0 – the size of the object in **unit**s.
- CAN_ERRET_OBD_NOOBJECT – object does not exist in the object dictionary.
- CAN_ERRET_OBD_NOSUBIND – sub-index does not exist.

int16 client_read_object_dictionary(canindex index, cansubind subind, canbyte *data);
Reading an object from the client (master) communication profile. The result is converted into a byte format and located at ***data**. The byte order is not changed. The application must allocate a buffer large enough to hold the entire object. The size of the object can be determined using the function client_get_object_size(...).
Parameters:
- **index** – communication object index.
- **subind** – communication object sub-index.
- ***data** – byte pointer to the data.

Return values: normal completion = 0; error < 0.
- CAN_RETOK – normal completion.
- CAN_ERRET_OBD_NOOBJECT – object does not exist in the object dictionary.
- CAN_ERRET_OBD_NOSUBIND – sub-index does not exist.
- CAN_ERRET_OBD_WRITEONLY – attempt to read a write only object.

int16 client_write_object_dictionary(canindex index, cansubind subind, canbyte *data);
Writing an object to the client (master) communication profile. The function places the object located at ***data** in the object dictionary entry. The byte order is not changed. Before calling the function, the application must convert the data into a byte format.
Parameters:
- **index** – communication object index.
- **subind** – communication object sub-index.
- ***data** – byte pointer to the data.

Return values: normal completion = 0; error < 0.
- CAN_RETOK – normal completion.
- CAN_ERRET_OBD_NOOBJECT – object does not exist in the object dictionary.
- CAN_ERRET_OBD_NOSUBIND – sub-index does not exist.
- CAN_ERRET_OBD_READONLY – attempt to write a read only object.

int16 client_read_obd_u32(cannode node, canindex index, cansubind subind, unsigned32 *du32);
Reading an object up to 32 bits size. The function is to facilitate access to objects whose length does not exceed 32 bits. It reads an object dictionary entry, defined by the **node**, **index** and **subind**. The result is located at ***du32**. If **node** = 0, data is read from the client (master) communication profile, otherwise from the slave application profile, mapped in the master. The object size must not exceed 32 bits, otherwise the behavior of the function and the result is unpredictable.
Parameters:
see client_read_object_dictionary(...), node_read_manstan_objdict(...).
Return values: normal completion = 0; error < 0.
see client_read_object_dictionary(...), node_read_manstan_objdict(...).

int16 client_write_obd_u32(cannode node, canindex index, cansubind subind, unsigned32 du32);
Writing an object up to 32 bits size. The function is to facilitate access to objects whose length does not exceed 32 bits. It puts value **du32** to the object dictionary entry, defined by the **node**, **index** and **subind**. If **node** = 0, data is written to the client (master) communication profile, otherwise to the slave application profile, mapped in the master. The object size must not exceed 32 bits, otherwise the behavior of the function and the result is unpredictable.
Parameters:
see client_write_object_dictionary(...), node_write_manstan_objdict(...).
Return values: normal completion = 0; error < 0.
see client_write_object_dictionary(...), node_write_manstan_objdict(...).

int16 produce_emcy_default(unsigned16 errorcode);
Generates the master emergency message. The message is logged, but not transmitted to the CAN network.
Parameters:
- **errorcode** – emergency error code.

Return values: normal completion = 0.
- CAN_RETOK – normal completion.

Slave functions API

canbyte *server_get_object_pointer(canindex index, cansubind subind);
Getting byte pointer to the slave device object.
Parameters:
- **index** – slave object index.
- **subind** – slave object sub-index.

Return values:
- not NULL – byte pointer to the object defined by the function arguments.
- NULL – the object is not accessible through a pointer.

int16 server_see_access(canindex index, cansubind subind);
Getting access bit mask to the slave device object.
Parameters:
- **index** – slave object index.
- **subind** – slave object sub-index.

Return values: access bit mask >= 0; error < 0.
- CAN_MASK_ACCESS_PDO – PDO mapping allowed for this object (LSB = 1).
- CAN_MASK_ACCESS_RO – read access for the object (LSB+1 = 1).
- CAN_MASK_ACCESS_WO – write access for the object (LSB+2 = 1).
- CAN_MASK_ACCESS_RW – read and write access (LSB+1 = 1 and LSB+2 = 1).
- CAN_ERRET_OBD_NOOBJECT – object does not exist in the object dictionary.
- CAN_ERRET_OBD_NOSUBIND – sub-index does not exist.

int32 server_get_object_size(canindex index, cansubind subind, int16 unit);
Object size request from the slave device profile. This function also detects the presence of the corresponding object in the dictionary. The size may be represented in bytes (**unit** = BYTES) or in bits (**unit** = BITS). The size in bits is used for bit-specific PDO mapping.
Parameters:
- **index** – slave object index.
- **subind** – slave object sub-index.
- **unit** – object size measurement unit: bytes (BYTES) or bits (BITS).

Return values: object size > 0; error < 0.
- > 0 – the size of the object in **unit**s.
- CAN_ERRET_OBD_NOOBJECT – object does not exist in the object dictionary.
- CAN_ERRET_OBD_NOSUBIND – sub-index does not exist.

int16 server_get_object_type(canindex index, cansubind subind);
Object type request from the slave device profile.
Parameters:
- **index** – slave object index.
- **subind** – slave object sub-index.

Return values: object type > 0; error < 0.
- > 0 – static data type index (0001_h to $001F_h$).
- CAN_ERRET_OBD_NOOBJECT – object does not exist in the object dictionary.
- CAN_ERRET_OBD_NOSUBIND – sub-index does not exist.

int16 server_read_object_dictionary(canindex index, cansubind subind, canbyte *data);
Reading an object from the slave device profile. The result is converted into a byte format and located at ***data**. The byte order is not changed. The application must allocate a buffer large enough to hold the entire object. The size of the object can be determined using the function

server_get_object_size(...).
Parameters:
- **index** – slave object index.
- **subind** – slave object sub-index.
- ***data** – byte pointer to the data.

Return values: normal completion = 0; error < 0.
- CAN_RETOK – normal completion.
- CAN_ERRET_OBD_NOOBJECT – object does not exist in the object dictionary.
- CAN_ERRET_OBD_NOSUBIND – sub-index does not exist.
- CAN_ERRET_OBD_WRITEONLY – attempt to read a write only object.

int16 server_read_obd_network(canindex index, cansubind subind, canbyte *data);
Reading an object from the slave device profile. The byte order is arranged in a network transfer sequence. If CAN_BYTE_ORDER = NORMAL the function fully corresponds to server_read_object_dictionary(...). When CAN_BYTE_ORDER = REVERSE the byte order for the basic numeric data types is reversed.
Parameters:
see server_read_object_dictionary(...).
Return values: normal completion = 0; error < 0.
see server_read_object_dictionary(...).

int16 server_write_object_dictionary(canindex index, cansubind subind, canbyte *data);
Writing an object to the slave device profile. The function places the object located at ***data** in the object dictionary entry. The byte order is not changed. Before calling the function, the application must convert the data into a byte format.
Parameters:
- **index** – slave object index.
- **subind** – slave object sub-index.
- ***data** – byte pointer to the data.

Return values: normal completion = 0; error < 0.
- CAN_RETOK – normal completion.
- CAN_ERRET_OBD_NOOBJECT – object does not exist in the object dictionary.
- CAN_ERRET_OBD_NOSUBIND – sub-index does not exist.
- CAN_ERRET_OBD_READONLY – attempt to write a read only object.

int16 server_write_obd_network(canindex index, cansubind subind, canbyte *data);
Writing an object to the slave device profile. The byte order is arranged in a local application sequence. If CAN_BYTE_ORDER = NORMAL the function fully corresponds to server_write_object_dictionary(...). When CAN_BYTE_ORDER = REVERSE the byte order for the basic numeric data types is reversed.
Parameters:
see server_write_object_dictionary(...).
Return values: normal completion = 0; error < 0.
see server_write_object_dictionary(...).

int16 server_read_obd_u32(canindex index, cansubind subind, unsigned32 *du32);
Reading slave profile object up to 32 bits size. The function is to facilitate access to objects whose length does not exceed 32 bits. It reads an object dictionary entry, defined by the **index** and **subind**. The result is located at ***du32**. The object size must not exceed 32 bits, otherwise the behavior of the function and the result is unpredictable.
Parameters:

see server_read_object_dictionary(...).
Return values: normal completion = 0; error < 0.
see server_read_object_dictionary(...).

int16 server_write_obd_u32(canindex index, cansubind subind, unsigned32 du32);
Writing slave profile object up to 32 bits size. The function is to facilitate access to objects whose length does not exceed 32 bits. It puts value **du32** to the object dictionary entry, defined by the **index** and **subind**. The object size must not exceed 32 bits, otherwise the behavior of the function and the result is unpredictable.
Parameters:
see server_write_object_dictionary(...).
Return values: normal completion = 0; error < 0.
see server_write_object_dictionary(...).

int16 produce_emcy(unsigned16 errorcode, unsigned16 addinf, canbyte *mserr);
Generates an emergency message with full information about the error. Sets in the pre–defined error field (object 1003_h) error code and additional information. Then sends the emergency message with the **errorcode**, the current state of the error register and manufacturer-specific error code ***mserr** (the first five bytes used). Emergency object must be valid and not inhibited.
Parameters:
- **errorcode** – emergency error code.
- **addinf** – pre-defined error field additional information.
- ***mserr** – manufacturer-specific error code (5 bytes).

Return values: normal completion = 0; error < 0.
- CAN_RETOK – normal completion.
- CAN_ERRET_EMCY_INVALID – Emergency object is invalid.
- CAN_ERRET_EMCY_INHIBIT – Emergency is inhibited.
- CAN_ERRET_NODE_STATE – the CAN node is in stropped of initializing state.
- CAN_ERRET_COMM_SEND – failed to send frame in the CAN network.

int16 produce_emcy_default(unsigned16 errorcode);
Creates an emergency message with minimal information about the error. Only error code is used. Additional information and manufacturer-specific error code are missing (reset to zero).
Parameters:
- **errorcode** – emergency error code.

Return values: normal completion = 0; error < 0.
see produce_emcy(...).

void clear_error_register(unsigned8 mask);
Bitwise clearing of the error register (object 1001_h). Clears error register bits for which the **mask** is set to 1. Generic error (bit 0) is reset only when all other errors are zero. In this case an urgent message with zero emergency code is issued (error reset or no error). Error codes in the range 1000_h to $10FF_h$ set only generic error (bit 0) of the error register which, in the absence of other errors, will be reset with any **mask** value.
Parameters:
- **mask** – bit mask.

int16 get_flash_nodeid();
Reading CAN node-ID value from non-volatile memory.
Return values: CAN node-ID >= 0; error < 0.
- CAN_ERRET_FLASH_DATA – data in non-volatile memory is erroneous or inconsistent.

- CAN_ERRET_FLASH_VALUE – a value is not recorded in non-volatile memory.

int16 get_flash_bitrate_index();
Reading CAN bit rate index from non-volatile memory.
Return values: bit rate index >= 0; error < 0.
see get_flash_nodeid().

int16 put_flash_nodeid(cannode node);
Store CAN node-ID value in non-volatile memory.
Parameters:
- **node** – storable node-ID.

Return values: normal completion = 0; error < 0.
- CAN_RETOK – normal completion.
- CAN_ERRET_FLASH_INIT – non-volatile memory initializing error.
- CAN_ERRET_FLASH_DATA – the data recorded in non-volatile memory is erroneous or inconsistent.

int16 put_flash_bitrate_index(unsigned8 br);
Store CAN bit rate index in non-volatile memory.
Parameters:
- **br** – storable bit rate index.

Return values: normal completion = 0; error < 0.
see put_flash_nodeid(...).

User edited functions API

Final programming of these functions is carried out depending on the requirements of the target application.

unsigned32 read_dev_type_object(canindex index, cansubind subind);
Specifies the description of the device type (object 1000_h), manufacturer status register (object 1002_h) and the identity object (1018_h). Modules \slave__can_device_*.h for different devices. Called when reading the corresponding object dictionary indexes.
Parameters:
- **index** – object dictionary index.
- **subind** – object dictionary sub-index.

Return values:
The corresponding object value.

void read_dev_string_object(canindex index, cansubind subind, canbyte *data);
Specifies a symbolic description of the device: manufacturer device name (object 1008_h), manufacturer hardware version (object 1009_h) and manufacturer software version (object $100A_h$). Modules \slave__can_device_*.h for different devices. Called when reading the corresponding object dictionary indexes.
Parameters:
- **index** – object dictionary index.
- **subind** – object dictionary sub-index.
- ***data** – byte pointer to the visible string, which is a symbolic description of the device.

cannode get_node_id(void);
Returns CANopen device node-ID. Module \common__can_init.c. For master application the function should return zero.
Return value:
- CAN device node-ID (1 to 127 or 255 for slave devices). Read from non-volatile memory, or set, for example, by switches.

unsigned8 get_bit_rate_index(void);
Returns CAN bit rate index. Module \common__can_init.c.
Return value:
- CAN bit rate index. Read from non-volatile memory, or set, for example, by switches.

unsigned32 get_serial_number(void);
Returns CANopen slave device serial number. Module \common__can_init.c.
Return value:
- CANopen slave device serial number (object $1018_h sub4_h$).

void consume_sync(unsigned8 sc);
Processes the SYNC object. Module \common__can_events.c. Called upon receipt of the synchronization object.
Parameters:
- **sc** – SYNC counter value (1 to 240).

void no_sync_event(void);
The sync message has not been received within the configured communication cycle period timeout (object 1006_h). Module \common__can_events.c. Provides at least the corresponding led

indication.

void consume_time(canframe *cf);
Processes the TIME object. Module \common__can_events.c. Called upon receipt of the time stamp object and can be used to correct the device local time.
Parameters:
- ***cf** – CAN frame containing the time stamp object in the format of the TIME_OF_DAY structure.

void consume_controller_error(canev ev);
CAN interface error signal handler. Module \common__can_events.c. Bus-off condition of the CAN controller is processed according to the settings of the object 1029_h (error behavior object). Handling of other errors provides the transfer of the corresponding emergency message and led indication. CAN controller error codes are defined in the CAN data link driver header file.
Parameters:
- **ev** (int16) – error code:
  CIEV_WTOUT – write timeout occurred,
  CIEV_EWL – error warning limit,
  CIEV_BOFF – bus off,
  CIEV_HOVR – hardware overrun,
  CIEV_SOVR – software overrun.

void pdo_activated_master(cannode node, canindex index, cansubind subind);
Reports master PDO activation. Module \common__can_events.c. The function informs application that the PDO object was successfully written to the slave application profile, mapped in the master.
Parameters:
- **node** – the slave node-ID.
- **index** – mapped application object index.
- **subind** – mapped application object sub-index.

void pdo_activated_slave(canindex index, cansubind subind);
Reports slave PDO activation. Module \common__can_events.c. The function informs application that the PDO object was successfully written to the slave device profile.
Parameters:
- **index** – slave object index.
- **subind** – slave object sub-index.

void master_emcy(unsigned16 errorcode);
The master emergency. Module \common__can_events.c. Called when the **errorcode** event appears in the master. The emergency message is not transmitted in the CAN network.
Parameters:
- **errorcode** – emergency error code.

void consume_emcy(canframe *cf);
Handles an emergency message. Module \common__can_events.c. Called upon receipt of the emergency message from any slave device.
Parameters:
- ***cf** – emergency CAN frame.

void can_client_state(struct sdocltappl *ca);
Information about the state of the SDO client transaction. Module \common__can_events.c. Reports the status after the transaction completion.
Parameters:
- ***ca** – the structure used by the SDO client when exchanging data with SDO protocol.

void heartbeat_event(cannode node);
Handles the heartbeat event with the state 'occurred' for the **node**. Module \common__can_events.c. Called if the heartbeat is not received within the heartbeat consumer time.
Parameters:
- **node** – NMT slave node-ID.

void heartbeat_resolved(cannode node);
Handles the heartbeat event with the state 'resolved' for the **node**. Module \common__can_events.c. Called when resuming receipt of the heartbeat messages.
Parameters:
- **node** – NMT slave node-ID.

void node_guarding_event(cannode node);
Handles the node guarding event with the state 'occurred' for the **node**. Module \common__can_events.c.
Parameters:
- **node** – NMT slave node-ID.

void node_guarding_resolved(cannode node);
Handles the node guarding event with the state 'resolved' for the **node**. Module \common__can_events.c.
Parameters:
- **node** – NMT slave node-ID.

void bootup_event(cannode node);
Handles the boot-up event for the **node**. Module \common__can_events.c.
Parameters:
- **node** – NMT slave node-ID.

void node_state_event(cannode node, canbyte state);
Logs the NMT **state** of the **node**, received using the heartbeat or node guarding protocol. Module \common__can_events.c. Called each time when receiving NMT state of any slave node.
Parameters:
- **node** – NMT slave node-ID.
- **state** – NMT state of the **node**.

void life_guarding_event(void);
Handles the life guarding event with the state 'occurred'. Module \common__can_events.c. Invoked if the NMT slave has not been polled during its lifetime.

void life_guarding_resolved(void);
Handles the life guarding event with the state 'resolved'. Module \common__can_events.c. Invoked if the NMT slave polling has resumed.

void no_pdo_event(canindex index);
An expected PDO has not been received before the event-timer elapsed. Module \common__can_events.c. Provides at least the corresponding emergency message generation and led indication.
Parameters:
- **index** – RPDO communication parameter index.

void can_timer_overlap(void);
The CANopen timer ticks overlap was registered. Module \common__can_events.c. Provides at least the corresponding emergency message generation.

void can_cache_overflow(canbyte state);
CANopen cache overflow. Module \common__can_events.c. Provides at least the error registration.
Parameters:
- **state** – NMT state of the CAN node.

void can_init_pdo_map(void);
The initialization of static PDO mapping. Editable PDO components are placed in the module \common\pdomapping__map__static.h. Only used in the CANopen slave, when the parameter CAN_PDO_MAPPING_MODE = STATIC. Specifies mapping of all received and transmitted PDOs in the static mapping mode.

General management functions API

void can_set_datalink_layer(unsigned8 mode);
Setting of logical access to the CAN network driver. Performs enabling and disabling of the CAN data link layer on write. Attempts to output data to the disconnected CAN network may cause significant delays due to the timeouts in the driver and in the CANopen library. NMT slave device is logically reconnected to the CAN network upon receipt of any intended NMT command.
Parameters:

- **mode** – logical access to the CAN network driver. ON – normal operation, all frames are transmitted to the CAN network. OFF – all pending and sent frames will be cancelled. Upon the library initialization normal mode (ON) is set.

System–dependent functions API

These functions are placed in the relevant modules of the CANopen root directory. Module __can_system.c is the system–dependent units manager.

void can_sleep(int32 microseconds);
Time delay function.
Parameters:
- **microseconds** – the time delay in microseconds. The exact time delay is determined by the resolution of the corresponding system timer. Any positive value of the parameter should provide a non-zero delay.

void can_init_system_timer(void (*handler)(void));
The CANopen timer initialization. The timer signal should have a high priority and to block other signals or the operating system threads during its own execution. The timer handler function is signal-safe and may be assigned directly to hardware interrupts, including not self-blocking. In the case when the timer is executed as a separate operating system thread, the OS manager may not guarantee the continuous execution of this thread. In this case, it is recommended to make the timer handler code (function background() of the can_backinit.c module) as a single critical section.
Parameters:
- **handler** – the timer handler function, has the prototype void background(void).

void can_cancel_system_timer(void);
Cancel the CANopen timer. Stops or shuts down the timer.

void init_critical(void);
void enter_critical(void);
void leave_critical(void);
Initialization, entry and exit of a critical section. The functions are designed to ensure the atomicity of the semaphore operations and the continuity of the code segments when using the library in a multithread environment. In this case the CANopen timer and CAN frames handler are run as separate threads. This situation occurs for example, when using Windows OS. The functions are implemented by using macros CAN_CRITICAL_INIT, CAN_CRITICAL_BEGIN and CAN_CRITICAL_END defined in the can_macros.h module. For single-thread applications (microcontrollers, operating systems that support signals) the library code provides the opportunity to work with non-atomic semaphores and the macros can be empty.

void enable_can_transmitter(void);
void disable_can_transmitter(void);
Unlock (enable) and lock (disable) the transmitting CAN transceiver. The functions are designed to prevent the false signals transmission in the CAN network at device power-on. The transceiver is unlocked while initializing the CAN subsystem (module can_backinit.c).

LED indication module

CANopen NMT slave device LED indication is implemented in accordance with the indicator specification CiA 303 part 3 v. 1.4. Either two LEDs ERROR (red) and RUN (green) or one bicolor (red/green) LED can be utilized. The led type is configured with the CAN_LED_INDICATOR parameter. In case, there is a conflict between turning the bicolor LED on green versus red, the LED is turned on red. For correct operation of the LEDs in all modes the CANopen timer period should not exceed 50000 microseconds (frequency greater than 20 Hz).

Error LED (red)

Error LED	Description
Off	No error. Red LED if turned OFF when the device receives any intended NMT command.
Flickering alternately with green LED	The auto-bitrate detection is in progress or LSS services are in progress.
Blinking	General configuration error.
Single flash	Warning limit reached. At least one of the error counters of the CAN controller has reached or exceeded the warning level (too many error frames).
Double flash	A guard event (NMT-slave or NMT-master) or a heartbeat event (heartbeat consumer) has occurred.
Triple flash	The sync message has not been received within the configured communication cycle period time out (object 1006_h).
Quadruple flash	An expected PDO has not been received before the event-timer elapsed.
On	The CAN controller is bus off.

Run LED (green)

Run LED	Description
Flickering alternately with red LED	The auto-bitrate detection is in progress or LSS services are in progress.
Blinking	The device is in state PRE-OPERATIONAL.
Single flash	The device is in state STOPPED.
Double flash	Reserved.
Triple flash	A software download is running on the device.
Quadruple flash	Reserved.
On	The device is in state OPERATIONAL.

Both LEDs turn off when the device receives invalid NMT command from the CANopen network, while its NMT state is not changed.

Physical LED control functions

Function stubs, which body has to be the call to the LED controlling registers.

void green_led_on(void);
Green LED physical ON.
void green_led_off(void);
Green LED physical OFF.
void red_led_on(void);
Red LED physical ON.
void red_led_off(void);
Red LED physical OFF.

LED functions API

void set_led_red_on(void);
void set_led_red_off(void);
Red LED ON and OFF.

void set_led_green_on(void);
void set_led_green_off(void);
Green LED ON and OFF.

void set_leds_flickering(void);
Red and Green LEDs flickering alternately.

void set_led_red_blinking(void);
Red LED blinking.

void set_led_green_blinking(void);
Green LED blinking.

void set_led_red_single_flash(void);
void set_led_red_double_flash(void);
void set_led_red_triple_flash(void);
void set_led_red_quadruple_flash(void);
Red LED single, double, triple and quadruple flash respectively.

void set_led_green_single_flash(void);
void set_led_green_double_flash(void);
void set_led_green_triple_flash(void);
void set_led_green_quadruple_flash(void);
Green LED single, double, triple and quadruple flash respectively.

Library application examples

The CANopen library test application examples are given in the modules:

- \master__can_test_application.c – client operations and slave test profile object dictionary mapping.
- \slave__obdms_slave_test.h – object dictionary example for the slave test profile.
- \master__obdms_master_test.h – object dictionary mapping example for the slave test profile.

All functions of these modules are commented in detail.

CAN node-ID and bit rate index

CAN node-ID

Node-ID	Usage
1 to 127	Valid CANopen node-IDs.
255	Not configured CANopen device.

Standard CiA bit timing parameter table

The table_selector for /CiA301/ bit timing table is 0 (zero).
Definition of table_index for the table.

table_index	Bit rate
0	1000 kbit/s
1	800 kbit/s
2	500 kbit/s
3	250 kbit/s
4	125 kbit/s
5	reserved
6	50 kbit/s
7	20 kbit/s
8	10 kbit/s
9	Automatic bit rate detection

CANopen error codes

SDO abort codes

Abort code	Description
0503 0000_h	Toggle bit not alternated.
0504 0000_h	SDO protocol timed out.
0504 0001_h	Client/server command specifier not valid or unknown.
0504 0002_h	Invalid block size (block mode only).
0504 0003_h	Invalid sequence number (block mode only).
0504 0004_h	CRC error (block mode only).
0504 0005_h	Out of memory.
0601 0000_h	Unsupported access to an object.
0601 0001_h	Attempt to read a write only object.
0601 0002_h	Attempt to write a read only object.
0602 0000_h	Object does not exist in the object dictionary.
0604 0041_h	Object cannot be mapped to the PDO.
0604 0042_h	The number and length of the objects to be mapped would exceed PDO length.
0604 0043_h	General parameter incompatibility reason.
0604 0047_h	General internal incompatibility in the device.
0606 0000_h	Access failed due to a hardware error.
0607 0010_h	Data type does not match, length of service parameter does not match.
0607 0012_h	Data type does not match, length of service parameter too high.
0607 0013_h	Data type does not match, length of service parameter too low.
0609 0011_h	Sub-index does not exist.
0609 0030_h	Invalid value for parameter (download only).
0609 0031_h	Value of parameter written too high (download only).
0609 0032_h	Value of parameter written too low (download only).
0609 0036_h	Maximum value is less than minimum value.
060A 0023_h	Resource not available: SDO connection.
0800 0000_h	General error.
0800 0020_h	Data cannot be transferred or stored to the application.
0800 0021_h	Data cannot be transferred or stored to the application because of local control.
0800 0022_h	Data cannot be transferred or stored to the application because of the present device state.
0800 0023_h	Object dictionary dynamic generation fails or no object dictionary is present.
0800 0024_h	No data available.

Emergency error code classes

Error code	Description
$00xx_h$	Error reset or no error.
$10xx_h$	Generic error.
$20xx_h$	Current.
$21xx_h$	Current, CANopen device input side.
$22xx_h$	Current inside the CANopen device.
$23xx_h$	Current, CANopen device output side.
$30xx_h$	Voltage.
$31xx_h$	Mains voltage.
$32xx_h$	Voltage inside the CANopen device.
$33xx_h$	Output voltage.
$40xx_h$	Temperature.
$41xx_h$	Ambient temperature.
$42xx_h$	CANopen device temperature.
$50xx_h$	CANopen device hardware.
$60xx_h$	CANopen device software.
$61xx_h$	Internal software.
$62xx_h$	User software
$63xx_h$	Data set.
$70xx_h$	Additional modules.
$80xx_h$	Monitoring.
$81xx_h$	Communication.
$82xx_h$	Protocol error.
$90xx_h$	External error.
$F0xx_h$	Additional functions.
$FFxx_h$	CANopen device specific.

Emergency error codes

Error code	Description
0000_h	Error reset or no error.
1000_h	Generic error.
2000_h	Current – generic error.
2100_h	Current, CANopen device input side – generic.
2200_h	Current inside the CANopen device – generic.
2300_h	Current, CANopen device output side – generic.

3000_h	Voltage – generic error.
3100_h	Mains voltage – generic.
3200_h	Voltage inside the CANopen device – generic.
3300_h	Output voltage – generic.
4000_h	Temperature – generic error.
4100_h	Ambient temperature – generic.
4200_h	Device temperature – generic.
5000_h	CANopen device hardware – generic error.
6000_h	CANopen device software – generic error.
6100_h	Internal software – generic.
6180_h	CANopen cache overflow.
6190_h	CANopen timer initialization error.
6191_h	CANopen timer overlap.
$61A0_h$	Non-volatile memory data invalid (CRC error).
$61A1_h$	Non-volatile memory operations error.
6200_h	User software – generic.
6300_h	Data set – generic.
7000_h	Additional modules – generic error.
8000_h	Monitoring – generic error.
8100_h	Communication – generic.
8110_h	CAN overrun (objects lost).
8120_h	CAN in error passive mode.
8130_h	Life guard error or heartbeat error.
8140_h	Recovered from bus off.
8150_h	CAN-ID collision.
8180_h	CAN controller event «hardware overrun».
8181_h	CAN controller event «software overrun».
8182_h	CAN controller event «error warning limit».
8183_h	CAN controller event «write timeout».
8200_h	Protocol error – generic.
8210_h	PDO not processed due to length error.
8220_h	PDO length exceeded.
8230_h	DAM MPDO not processed, destination object not available.
8240_h	Unexpected SYNC data length.
8250_h	RPDO timeout.
9000_h	External error – generic error.

$F000_h$	Additional functions – generic error.
$FF00_h$	Device specific – generic error.

Color marked are manufacturer-specific error codes.
Errors 6180_h, 6190_h, $61A0_h$ and $61A1_h$ are recorded in the pre-defined error field (object 1003_h), but do not initiate EMCY, because the Emergency service can not be executed.

Generic pre-defined connection set

Broadcast objects

CAN-ID	Communication object	Object index
0	NMT	---
128 (80_h)	SYNC	1005_h
256 (100_h)	TIME	1012_h

Peer-to-peer objects

CAN-ID	Communication object	Object index
129 (81_h) – 255 (FF_h)	EMCY for CAN node-IDs 1 to 127	1014_h
385 (181_h) – 511 ($1FF_h$)	TPDO 1 for CAN node-IDs 1 to 127	1800_h
513 (201_h) – 639 ($27F_h$)	RPDO 1 for CAN node-IDs 1 to 127	1400_h
641 (281_h) – 767 ($2FF_h$)	TPDO 2 for CAN node-IDs 1 to 127	1801_h
769 (301_h) – 895 ($37F_h$)	RPDO 2 for CAN node-IDs 1 to 127	1401_h
897 (381_h) – 1023 ($3FF_h$)	TPDO 3 for CAN node-IDs 1 to 127	1802_h
1025 (401_h) – 1151 ($47F_h$)	RPDO 3 for CAN node-IDs 1 to 127	1402_h
1153 (481_h) – 1279 ($4FF_h$)	TPDO 4 for CAN node-IDs 1 to 127	1803_h
1281 (501_h) – 1407 ($57F_h$)	RPDO 4 for CAN node-IDs 1 to 127	1403_h
1409 (581_h) – 1535 ($5FF_h$)	SDO server->client for CAN node-IDs 1 to 127	1200_h
1537 (601_h) – 1663 ($67F_h$)	SDO client->server for CAN node-IDs 1 to 127	1200_h
1793 (701_h) – 1919 ($77F_h$)	NMT error control for CAN node-IDs 1 to 127	1016_h, 1017_h

Other objects

CAN-ID	Communication object
2020 ($7E4_h$)	LSS slave device messages
2021 ($7E5_h$)	LSS master device messages

Restricted CAN-IDs

The restricted CAN-ID shall not be used by any configurable communication object, neither for SYNC, TIME, EMCY, PDO or SDO.

CAN-ID	Communication object
0	NMT
1 (001_h) – 127 ($07F_h$)	reserved
257 (101_h) – 384 (180_h)	reserved
1409 (581_h) – 1535 ($5FF_h$)	default SDO server->client
1537 (601_h) – 1663 ($67F_h$)	default SDO client->server

1760 ($6E0_h$) – 1791 ($6FF_h$)	reserved
1793 (701_h) – 1919 ($77F_h$)	NMT error control
1920 (780_h) – 2047 ($7FF_h$)	Reserved

CANopen conformance test

CANopen conformance test plan (CiA 310) is designed to test devices utilizing the CANopen protocol. The device is tested for CiA 301 compliance as CAN network node. Its internal logic (application profile) is not verified. Any device that supports the CANopen protocol, must be approved (certified) using the conformance test.
CANopen conformance test software is distributed by the CAN in Automation. For CAN network access the conformance test uses a standardized set of functions, COTI (CANopen Test Interface). CAN interface manufacturer must provide COTI library for its CAN adapters.

CANopen conformance test implements the following operations:
- Check the device electronic data sheet (EDS) for compliance with the CiA 306.
- Testing network protocol according to the CiA 301. Generic pre-defined connection set with 11-bit CAN-IDs is used.
- Verification of compliance of the device object dictionary to its electronic specification (EDS).

In 2013 CiA released the third major version of the CANopen conformance test that supports the updated CiA standards. In some cases, the new version of the test performs more rigorous checking of CANopen protocols and the device object dictionary. Marathon CANopen library v. 2.3 and later was adapted for passing the conformance test of the third version.