CANopen Library Toolset

CANopen SW Library – Software User Manual

CAN-N7S-CANDP-SUM rev. 2.3

N7 SPACE SP. Z O.O.

Prepared by	Date and Signature
Konrad Grochowski	Konrad Grochowski 2025.09.08 17:28:22 +02'00'
Verified by	
Mateusz Dyrdół	Mateusz Dyrdół 2025.09.09 14:45:51 +02'00'
Approved by	
Seweryn Ścibior	Seweryn Ścibior Digitally signed by Seweryn Ścibior Date: 2025.09.09 16:20:28 +02'00'

ESA Contract No. 4000143237/23/NL/AS

Table of Contents

ESA Contract No. 4000143237/23/NL/AS

ESA Contract No. 4000143237/23/NL/AS

Change Record

Issue	Date	Change
1.0	2021-06-28	Initial release
1.1	2021-10-15	Fixes for CDR RIDs: • 7.2 – reworded sentences for better readability, • 11.3.6 – extended *venv* usage instruction • Captions added to all code listings • Removed outdated reference to *librt* • Explicit mention the Ubuntu 20.04 as the reference system • Added chapter 11.2.1 – Obtaining the source • Added 11.2.2.1 chapter about SCET and SUTC types • Added 11.2.2.2 chapter about cross-compilation
1.2	2021-11-18	Updated referenced documents' versions (for v3.1.3)
1.3	2021-11-26	Updated referenced documents' versions (for v3.2.0)
2.0	2024-10-03	CANopen Library Toolset project PDR: • Document identifier changed from CAN-N7S-UM-21001 to CAN-N7S-CANDP-SUM • New ESA contract identifier added to footer • Introduction updated • Reference documents updated • 10.6.3 Assertions and logging chapter added
2.1	2024-11-27	Release for MTR: • Reference documents updated
2.2	2025-05-29	Release for TRR: • Reference documents updated • Chapter 11.2.5 (endianness) added • Chapter 11.2.6 (MPLAB) added • Added compilation options for GR712RC
2.3	2025-09-08	Release for CDR/QR: • Reference documents updated • Examples updated to reference v3.5.0

ESA Contract No. 4000143237/23/NL/AS

1 Introduction

This document provides Software User Manual for the CANopen SW Library deliverable of the CANopen Library Toolset project.

CANopen SW Library (CANSW) is an adaptation to space industry requirements of an existing and field-tested open-source CANopen library (*lely-core*). CANSW is compliant with space-specific CANopen extensions defined in ECSS-E-ST-50-15C and ECSS Criticality Category B software requirements. It was developed in the scope of previous ESA activity and validated on representative hardware platform (SAMV71). In the scope of this project its validation will be extended to include other ARM (SAMRH71 and SAMRH707) and LEON3 (GR712RC) platforms.

Additionally in the scope of the activity CANopen Library Test Environment, Test Suite and Development Support Software will be developed.

CANopen Library Test Environment (CTESW) defines the environment required to execute CANopen Library Test Suite (CTSSW) which is used to validate CANSW. CTSSW was developed in the scope of previous ESA activity and is available as open-source software. In the scope of this project CTESW will be extended to support new platforms and CTSSW will be executed on those.

CANopen Library Development Support Software (CDSSW) is a set of new tools developed in the scope of this project and aiming at supporting design of CANopen networks using CANSW. It will provide user with capabilities to verify semantic correctness of the multiple nodes building the CANopen network and offer support with editing, monitoring and instrumenting of the network.

The Software User Manual is produced as a standalone document and structured according to the SUM Document Requirements Definition (DRD) given in Annex H of ECSS-E-ST-40C [AD1].

2 Applicable and reference documents

2.1 Applicable documents

ID	Title	Reference	Rev.
AD1	ECSS – Space engineering Software	ECSS-E-ST-40C	6 March 2009
AD2	ECSS – CANbus extension protocol	ECSS-E-ST-50-15C	1 May 2015

2.2 Reference documents

ID	Title	Reference	Rev.
RD1	CAN in Automation – CANopen application layer and communication profile	CiA 301	Version 4.2.0
RD2	CAN in Automation – Electronic data sheet specification for CANopen	CiA 306	Version 1.3.0
RD3	CANopen Library Toolset CANopen SW Library – Interface Control Document	CAN-N7S-CANDP-ICD	2.4
RD4	CANopen Library Toolset CANopen SW Library – Software Configuration File	CAN-N7S-CANDP-SCF	2.4
RD5	CANopen Library Toolset CANopen SW Library – Software Design Document	CAN-N7S-CANDP-SDD	2.4
RD6	CANopen Library Toolset CANopen SW Library – Software Requirements Specification	CAN-N7S-CANDP-SRS	2.3
RD7	CANopen Library Toolset CANopen SW Library – Failure Modes and Effects Analysis	CAN-N7S-CANDP-FMEA	2.3
RD8	CANopen Library Toolset Test Suite – Software User Manual	CAN-N7S-CTSDP-SUM	2.2

ESA Contract No. 4000143237/23/NL/AS

3 Terms, definitions and abbreviated terms

This document acronyms and abbreviations are listed here under.

CAN	Controller Area Network
CANDP	CANopen SW Library Data Package
CANSW	CANopen SW Library
CDSDP	CANopen Development Support Data Package
CDSSW	CANopen Development Support Software
CTESW	CANopen Test Environment Software
CTSDP	CANopen Test Suite Data Package
CTSSW	CANopen Test Suite Software
HWTB	Hardware Test Bench
N7S	N7 Space

ESA Contract No. 4000143237/23/NL/AS

4 Conventions

This Software User Manual describes a software project, therefore it refers to various commands that can be executed in the terminal and it presents various source code fragments. In order to make those special blocks more readable, numerous style conventions are used. This chapter quickly summarizes said conventions.

Short commands and code fragments that are embedded inside normal text paragraphs use `this style with a monospace font`.

Commands that are a bit longer or span multiple lines follow the following style:

```
$ command
Output (optional)
```

All commands listed in this manual were prepared and validated on Ubuntu 20.04 system. Any similar Linux system should support all of the commands, it is recommended to use Ubuntu/Debian family.

Directory contents listings follow the same convention:

```
include/
└── subfolder/
    └── file
lib/
└── a generic comment about contents of lib/
share/
```

Source code blocks use the below style:

```
co_nmt_t* nmt_service = co_nmt_create(network, device);
assert(nmt_service != NULL);  // must be non-null
```

The syntax highlighting colours used in the above block are defined as follows:

```
C Preprocessor directive
C Preprocessor include path
Type (built-in and user-created)
Function declaration and definition
Keywords
Variable definition
Struct member variable definition
NULL
String literal
Comments
Other
```

Blocks with DCF contents use the following scheme:

```
[Section]
Key=Value  # comment
```


ESA Contract No. 4000143237/23/NL/AS

5 Purpose of the Software

The main purpose of the Software is to provide a software stack supporting a subset of the CANopen protocol as defined in ECSS-E-ST-50-15C [AD2]. It provides an implementation of the Object Dictionary and the Network Management (NMT), Service Data Object (SDO), Process Data Object (PDO), Synchronization Object (SYNC) and Emergency Object (EMCY) protocols. It allows users to create programs that need to use the CANopen protocol to communicate with software on other devices, which are not necessarily using the same software stack. CANSW is a highly portable software library that can be used also on resource constrained bare-metal microcontrollers. The Software and its public Application Programming Interface are written in the C Programming Language (C99), which means it can also be used directly by software written in C++ and when using any other programming language that can interface with C, which is a very large set of languages. Even the C Standard Library is not required to be available for users of the software. The implementation is passive, it relies on the user to provide integration with the underlying CAN bus to send and receive CAN frames and to update the clock used by CANSW. That means that the implementation is independent from any specific CAN networking driver and system clocks.

CANSW includes also a Python tool – dcf2dev, that can optionally be used by the user to transform device configuration files (DCF - [RD2]) into C data structures.

Detailed software overview can be found is SRS [RD6] and SDD [RD5].

ESA Contract No. 4000143237/23/NL/AS

6 External view of the software

CANSW is delivered as an archive consisting of source files and autotools-based build system. The software itself consists of 4 libraries:

- liblely-compat – provides implementation of necessary parts of the C Standard Library
- liblely-util – provides implementation of data structures and various utilities
- liblely-can – provides implementation of base CAN network interfaces
- liblely-co – provides implementation of the CANopen protocol on top of the other libraries

The directory structure can be described as follows (for clarity reduced to most important items):

```
lely-core/
├── doc/
│   ├── Doxyfile.in - Doxygen configuration file
│   └── Makefile.am
├── docker/ - reference container configuration for developers (used by CI)
├── include/
│   ├── lely/
│   │   ├── can/
│   │   │   └── Contains header files of liblely-can
│   │   ├── co/
│   │   │   └── Contains header files of liblely-co
│   │   ├── compat/
│   │   │   └── Contains header files of liblely-compat
│   │   └── util/
│   │       └── Contains header files of liblely-util
│   └── Makefile.am
├── lib/
│   ├── can/
│   │   └── Contains Makefile.am and source code of liblely-can
│   ├── co/
│   │   └── Contains Makefile.am and source code of liblely-co
│   ├── compat/
│   │   └── Contains Makefile.am and source code of liblely-compat
│   ├── util/
│   │   └── Contains Makefile.am and source code of liblely-utils
│   └── Makefile.am
├── m4/
│   └── .m4 files used by the autotools build system
├── pkgconfig/
│   ├── .pc.in files with pkg-config metadata file templates
│   └── Makefile.am
├── python/
│   └── dcf-tools/
│       ├── dcf/
│       │   └── DCF file manipulation utility library for Python
│       ├── dcf2dev/
│       │   └── dcf2dev Python program sources
│       ├── Makefile.am
│       └── setup.py
```


ESA Contract No. 4000143237/23/NL/AS

```
├── unit-tests/
│   ├── can/
│   │   └── Contains Makefile.am and .cpp files with unit tests for liblely-can
│   ├── co/
│   │   └── Contains Makefile.am and .cpp files with unit tests for liblely-co
│   ├── compat/
│   │   └── Contains Makefile.am and .cpp files with unit tests for liblely-compat
│   ├── cpputest/
│   │   └── Contains Makefile.am and a .cpp file with sanity unit tests
│   ├── libtest/
│   │   └── Contains Makefiles and utility source code used by other unit tests
│   ├── util/
│   │   └── Contains Makefile.am and .cpp files with unit tests for liblely-util
│   └── Makefile.am
├── Makefile.am
└── configure.ac
```

The Software Configuration File [RD4] contains a detailed list of files in the library package along with their SHA-256 checksums.

ESA Contract No. 4000143237/23/NL/AS

7 Operations environment

7.1 General

The software in this project is designed to be included and used by other software. Only a C compiler is required to build the library, and a C++ compiler to build its unit tests. Apart from the library itself, there are no other software component requirements imposed on the final user software. The hardware parts used by the software are the Central Processing Unit (CPU), the Floating-Point Unit (FPU) and Random Access Memory (RAM). No other hardware, especially CAN specific, is required.

For reference, the software has been built in a continuous manner using the following environment.

Table 1 – CANSW build environment.

Tool	Version	Purpose
Container environment		
Docker	19.03.12	Container manager. Image containing all necessary build environment is one of the deliverables of the CANSW (build environment includes all other tools from this table).
Compilation environment		
gcc x86/x64	4.9.0 5.5.0 6.5.0 7.5.0 8.4.0 9.3.0 10.3.0 13.2.0	Supported GNU C Compiler versions for x86 compilation. Newest (10.x) version was used in the validation activities and is included in distributed Docker image. Other versions were verified by CANSW Continuous Integration system and unit-tested, but not validated.
gcc ARM	arm-gnu-toolchain-13.3-rel1-x86_64-arm-none-eabi	GNU C Compiler for target ARM platforms
XC32 ARM compiler	XC32 v4.45 8.3.1	Microchip XC32 compiler for SAM* microcontrollers.
bcc LEON3	sparc-gaisler-elf-gcc (bcc-v2.3.1) 13.2.1 20240119	BCC C Compiler (GCC compatible) for LEON3 platforms
Autotools	autoreconf 2.69	Build system
Unit-testing environment		
CppUTest	4.0	Unit test library

7.2 Hardware configuration

The software has been tested on both x86 and ARM hardware architectures. The CANopen protocol supports floating-point values, the FPU has to be enabled before using them. The software is designed to be incorporated into other user-specified software. Therefore the detailed hardware configuration is project dependent, while library itself is designed for portability. Because of that only a generic hardware and component deployment diagram can be provided.

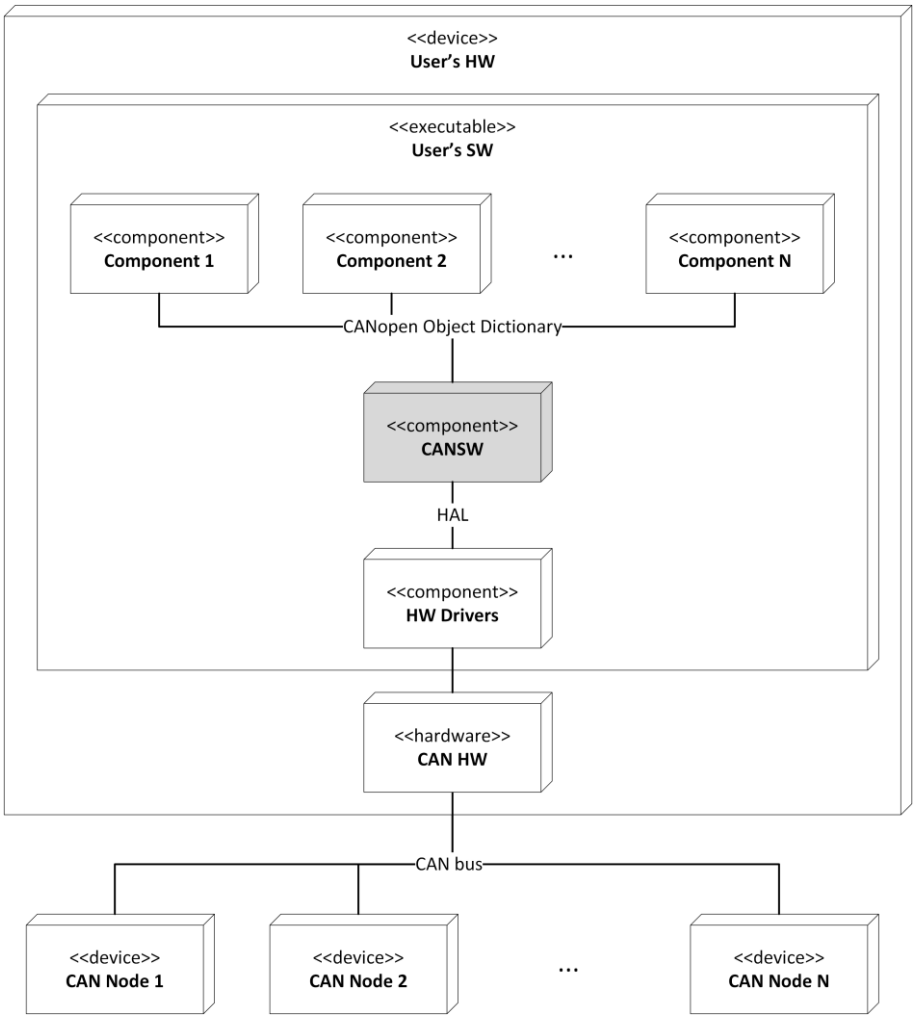


Figure 1 – CANSW generic deployment diagram.

7.3 Software configuration

The user should provide a Hardware Abstraction Layer (HAL) that will interface with the liblely-can library to provide clock and base CAN networking capabilities to the Software. Depending on the project this might involve integration with an underlying Operating System. The application code should interface with the liblely-co library for project-specific CANopen-based application logic. The CANopen device description also needs to be provided to the liblely-co library, either by compiling and linking manually written code with said description, or by using code generated by the dcf2dev tool from a Device Configuration File (DCF).

ESA Contract No. 4000143237/23/NL/AS

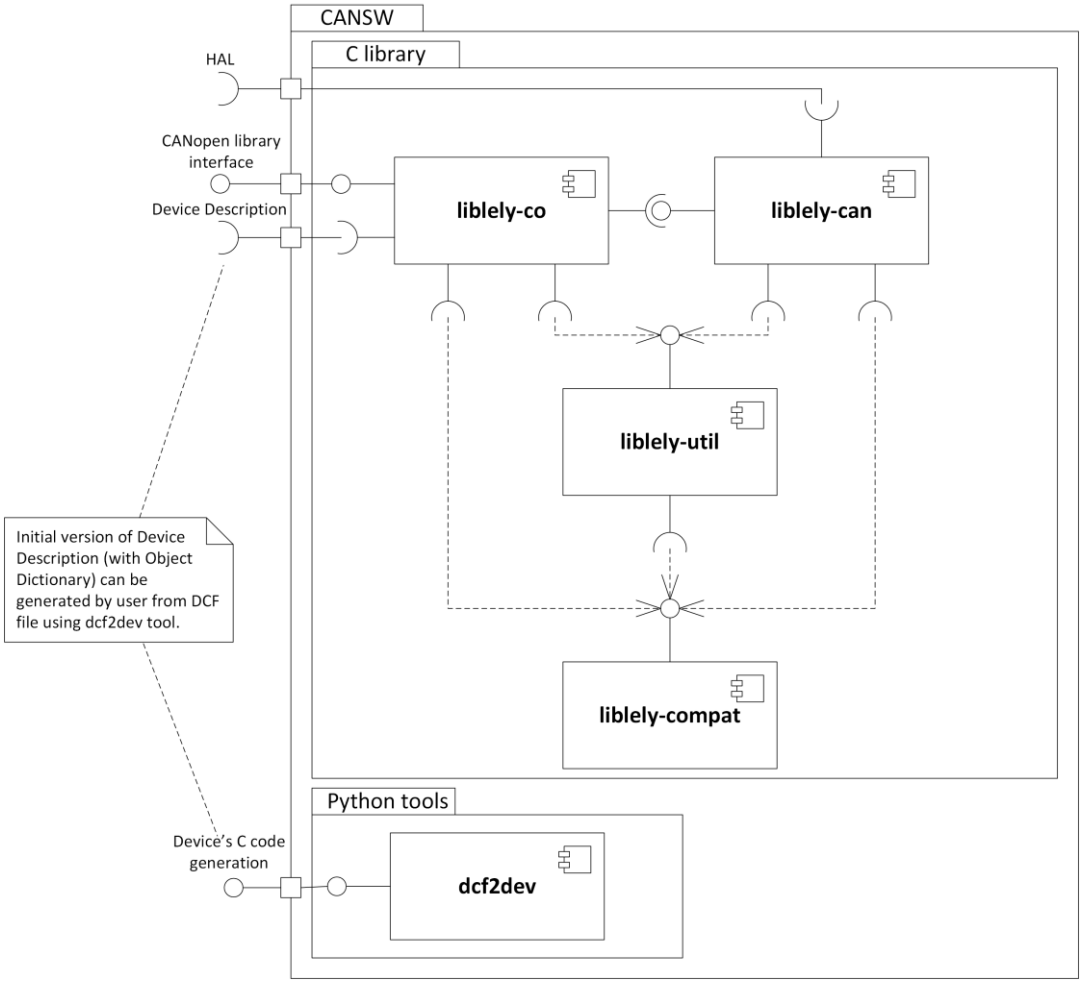


Figure 2 – CANSW components diagram.

7.4 Operational constraints

CANSW methods were not designed to be called directly from interrupt handlers and no special precautions were implemented in them. It is assumed in the analysis that events like "message received" are passed from interrupt handlers to HAL by the user and the Library code is executed in non-interrupt context. Note: methods *might* be safe to be used from interrupt handlers, but it is up the user to perform proper analysis.

CANSW is separated from HW and Operating System concerns and does not perform any internal synchronization to avoid data races. User should ensure that no CANSW methods are called from multiple threads/tasks on the same shared data or user should provide adequate synchronization techniques.

ESA Contract No. 4000143237/23/NL/AS

8 Operations basics

N/A - The software in this project is designed to be included and used by other software. Therefore there are no predefined operational tasks. Staffing concerns, standard daily operations and contingency operations are all dependent on the final software based on CANSW.

ESA Contract No. 4000143237/23/NL/AS

9 Operations manual

Operations manual is not provided for CANDP. Details related to operations of CTSSW (validation test suite) are provided in separate manual [RD8].

ESA Contract No. 4000143237/23/NL/AS

10 Reference manual

10.1 Introduction

A complete reference manual of the programming interfaces of each of the modules of CANSW is available as the `Doxygen`-generated documentation supplied with the [RD3] Annex A. It is generated from source code of the Library and inline comments written for every public API function. `Doxygen`-style comments in all public header files used for generation of the reference manual can also be inspected directly.

Commands listed in the following chapters assume Linux host – preferable Ubuntu 20.04 or similar. If user follows the environment setup from CTSDP SUM [RD8], all commands should be executed inside the Docker container (for example by preceding them with `docker-here` alias from [RD8]).

10.2 Help method

Each CANSW public function is documented with a basic description, the meaning of each input parameter and return value and a reminder on how to access error information in case of failure. This information is available in the `Doxygen`-generated documentation.

While building the Library, both the `configure` script and the generated `make` build system have built-in help describing available options:

```
$ ./configure --help
$ make --help
```

The `dcf2dev` tool also has a built-in help command.

```
$ dcf2dev --help
```

10.3 Screen definitions and operations

N/A

10.4 Commands and operations

N/A

10.5 Configuration options

10.5.1 Service tailoring

CANSW supports conditional compilation of CANopen service/protocol. This allows the user to control the object size by choosing to compile only the services that will be in fact used by the user code. This has to be specified at configuration time just before building the Library. All configuration options can be listed as already mentioned in [10.2] but the most important ones used in ECSS compliance mode and related to service tailoring are:

- CSDO, to disable pass `--disable-csdo`
- EMCY, to disable pass `--disable-emcy`

- SYNC, to disable pass `--disable-sync`
- RPDO, to disable pass `--disable-rpdo`
- TPDO, to disable pass `--disable-tpdo`
- NMT Master, to disable pass `--disable-master`

For example `$ ./configure --enable-ecss-compliance --disable-emcy --disable-csdo --disable-master` for a small build focused on PDO support.

The disabled services are reflected in the generated `config.h` configuration header that is used by the Library during build and can be then used by the user application. It contains C Preprocessor definitions for disabled services e.g.

```
#define LELY_NO_CO_CSDO=1
#define LELY_NO_EMCY=1
#define LELY_NO_CO_MASTER=1
```

for the example configuration above.

10.5.2 Built-in buffer sizes

CANSW, when configured in ECSS compliance mode, has disabled dynamic memory allocation support. In order for certain CANopen services to function, the implementation uses memory buffers with static predefined sizes. The user can overwrite the default sizes if the need arises. Below is the table of C Preprocessor definitions that control various buffers and their default values. To make sure that the Library and user code have consistent data structure definitions, overwritten values should be set when building both CANSW and user code.

Table 2 – C Preprocessor configuration variables.

C Preprocessor definition name	Default value	Description
`CO_EMCY_CAN_BUF_SIZE`	16	The maximum number of EMCY messages to send pending on inhibit timer
`CO_EMCY_MAX_NMSG`	8	The maximum number of EMCY errors in the error stack
`CO_ARRAY_CAPACITY`	256	The maximum size (in bytes) of a CANopen array value
`CO_SDO_REQ_MEMBUF_SIZE`	8	The size in bytes of an SDO upload/download request memory buffer, default large enough to accommodate basic data types
`CO_SSDO_MEMBUF_SIZE`	127 * 7	The maximum size (in bytes) of Server SDO memory buffer for incoming data, default large enough to accommodate maximum block size used by SDO block transfer
`CO_CSDO_MEMBUF_SIZE`	8	The maximum size (in bytes) of Client SDO memory buffer for incoming data, default large enough to accommodate basic data types
`CO_NMT_CAN_BUF_SIZE`	16	The maximum number of CAN frames used by NMT Master for buffered requests to NMT slaves
`CO_NMT_MAX_NHB`	127	The maximum number of NMT Heartbeat consumers, default equal to maximum number of CANopen nodes

10.6 Error messages

Please refer to FMEA [RD7] Table 4 for details about error messages and failure modes that can occur while accomplishing any of the user's functions, including the meaning of each message and recommended action to be taken after.

10.6.1 Error codes

In case of calling a function documented as setting an error code or number, the currently set error code can be obtained using the `get_errnum()` function available in `<lely/util/error.h>` header file. The Library itself uses a handful of error codes described below. More error codes are available for use in the user software, all of them are listed and documented in the `Doxygen` documentation for `<lely/util/error.h>`.

Table 3 – Error codes used internally by CANSW

Error code constant	Description
`ERRNUM_INVAL`	Invalid argument provided
`ERRNUM_NOMEM`	Not enough memory available
`ERRNUM_PERM`	Operation not permitted
`ERRNUM_NOSYS`	Function not supported
`ERRNUM_AGAIN`	Try again

In order to reset the currently set error number, after fixing the issue, one should execute `set_errnum(ERRNUM_SUCCESS);`.

A custom error code storing and reading functionality can be implemented by the user by calling `set_errc_set_handler` and `get_errc_set_handler`. One use case for changing the default behaviour is to use a custom storage or synchronization mechanism of the possibly shared error state. Please refer to the `Doxygen` documentation for more details.

10.6.2 Abort codes

In CANopen SDO transfers, abort codes are used to communicate an error between Client SDO and Server SDO services. CANSW provides C Preprocessor constant definitions of SDO abort codes as defined in [RD1] Table 22 in the `<lely/co/sdo.h>` header file. All abort code definitions follow the same naming scheme e.g. `CO_SDO_AC_PARAM_RANGE`. Please refer to [RD1] and the `Doxygen` documentation of the mentioned header file.

Client SDO service API provides indication and confirmation callback functions that can be set to monitor request progress and receive abort codes from Server SDO. For details refer to the `Doxygen` documentation of `co_csdo_set_dn_ind`, `co_csdo_set_up_ind`, `co_csdo_dn_con_t` and `co_csdo_up_con_t`.

Applications based on the Server SDO service can provide specific abort code handling logic by setting user-provided upload and download indication functions using `co_obj_set_dn_ind`, `co_sub_set_dn_ind`, `co_obj_set_up_ind` and `co_sub_set_up_ind` CANSW function. Refer to `Doxygen` documentation for details.

10.6.3 Assertions and logging

By default the library is built with support for assertions in the code. This is a recommended option with initial deployment of the library – assertions help to detect misuse of the C API. When necessary for performance / code size, they can be disable using standard C compilation define `NDEBUG`.

On embedded environments (especially bare-metal) to access the information provided by the assertion user will need to implement compiler specific assertion handling procedure. For GCC compiler it is usually called `__assert_func` and receives all assertion data as arguments. Those can be logged or discarded – it is up for the user to decide. Minimal approach would be to provide empty method as a location for inserting break point for debugging.

Additionally CANSW supports human-readable logging. This requires support for dynamic allocation and hence cannot be used in the "ECSS compliant" version. But if such needs arise, for debugging purposes, the library can be built without `-enable-ecss-compliance` and with `--enable-diag` option. Messages will be provided to standard output and it is up to the user to configure the system in such way, that the output will be visible.

ESA Contract No. 4000143237/23/NL/AS

11 Tutorial

11.1 Introduction

This tutorial serves as an introduction to the CANopen SW Library. Its goal is to demonstrate how to use the provided API to perform basic tasks related to the CANopen protocol. It covers building the Library in the ECSS compliance mode, assuming a Linux-based environment with embedded system characteristics i.e. with no dynamic memory allocation. For simplicity and readability, code fragments in this tutorial make use of the C Standard Library, but the built and the Library do not, due to the ECSS compliance mode. It also covers providing and using a custom memory allocator, example integration with a base CAN networking stack and an external clock. It covers CANopen specific topics like the Object Dictionary, NMT Master and Slave nodes, basic PDO and SDO transfer services.

This tutorial assumes a basic level of knowledge of the CANopen protocol and only provides an introduction to the Library, written specifically for software engineers – potential users of the Library.

11.2 Getting started

11.2.1 Obtaining the source

CANSW source can be obtained by extracting delivered ZIP archive as in Listing 1.

Listing 1 – Unpacking CANSW source from ZIP file.

```
$ unzip CAN-CANDP-library-src-v3_5_0.zip  # assuming version 3.5.0
```

Or (recommended option on Linux as CANSW uses symbolic-links) from TAR BZIP2 - Listing 2.

Listing 2 – Unpacking CANSW source from TAR BZIP2 file (recommended for Linux).

```
$ tar -xvf CAN-CANDP-library-v3_5_0.tar.bz2  # assuming version 3.5.0
```

Alternatively CANSW source can be accessed using publicly available code repository by executing the commands from Listing 3 (assuming version 3.5.0 of the CTSSW).

Listing 3 – Retrieving CANSW source from GitLab.com repository.

```
$ git clone https://gitlab.com/n7space/canopen/lely-core.git --depth=1 --branch=v3.5.0
```

11.2.2 Building the Library

This section assumes that the Library's main directory is the current working directory. In order to build a static version of the library in “debug” configuration, commands from Listing 4 should be executed.

Listing 4 – Build tool configuration for building library in *debug* mode (without optimalization).

```
$ autoreconf -i
$ CFLAGS="-O0 -g" CXXFLAGS="-O0 -g" ./configure --disable-shared --enable-ecss-compliance
$ make
```

To build an optimized "release" variant of the library the `CFLAGS` and `CXXFLAGS` environment variables should not exist or should be set to `-O2 -g`, which will be assumed as a default in the former case (see Listing 5).

Listing 5 – Build tool configuration for building library in *release* mode (with optimization).

```
$ unset CFLAGS
$ unset CXXFLAGS
$ ./configure --disable-shared --enable-ecss-compliance
$ make
```

In order to see more configuration options, execute the command from Listing 6.

Listing 6 – Build configuration *help* command.

```
$ ./configure --help
```

Unit tests require the `CppUTest` library. The `configure` script uses `pkg-config` to find it. In case of a custom manual build of the dependency, user might need adjusting the `PKG_CONFIG_PATH` environment variable. Both unit tests and the `CppUTest` library require a C++ compiler. The library itself requires only a C compiler.

To build the library and execute the unit tests, use command from Listing 7.

Listing 7 – Command for building the library and executing unit-tests.

```
$ make check
```

Note: unit tests require a full ECSS-compliant build of the Library, service tailoring is not supported by them. See [10.5.1].

To build the `Doxygen` documentation one has to have `Doxygen` and `Graphviz` installed on the development system at configure time (during execution of `./configure` command, like in Listing 4 or Listing 5). Then the command from Listing 8 will build the documentation in HTML format and make it available at `doc/html/index.html`.

Listing 8 – Command for generating documentation from the code.

```
$ make html
```

The documentation should be built and available at `doc/html/index.html`.

11.2.2.1 SCET and SUTC time types support

The Library offers optional support for SCET and SUTC time types. To enable those non-standard CANopen types, user needs to provide data type to be used to identify those values in the protocol. This has to be done using C language defines passed via `CPPFLAGS`:

- `CO_DEFTYPE_TIME_SCET`
- `CO_DEFTYPE_TIME_SUTC`

Listing 9 presents an example which enables SUTC type and assigns `0x0061` type identifier to it during Library build configuration. User has to be aware, that such setting of `CPPFLAGS` overwrites the default one, so special care might be required to ensure proper set of all required flags.

Listing 9 – Build tool configuration for building library with SUTC type support enabled.

```
$ CPPFLAGS="-DCO_DEFTYPE_TIME_SUTC=0x0061" ./configure --enable-ecss-compliance
```


ESA Contract No. 4000143237/23/NL/AS

11.2.2.2 Cross-compilation configuration

Build tool support cross-compilation – building the Library for platform other than the host one. Assuming GNU compatible build toolchain is available in the system (*gcc, ar, libtool* etc.) and is prefixed with *platform-name* (e.g. *platform-name-gcc*), configuring build of the Library to use that tool can be done by passing `--host=platform-name` parameter to `configure` script.

User should be aware, that choosing the tool might not be enough for properly configuring build for selected platform. Subset of compilation and linking flags might require to be set up according to given platform requirements. Those flags include:

- `CFLAGS`
- `CPPFLAGS`
- `LDFLAGS`

It is recommended for embedded platforms to disable building of executables (especially for bare metal platforms), by passing `--disable-tests --disable-unit-tests` to the configuration.

User must take special care for merging various flags (compilation mode, cross-compilation specific options, etc.) and other customization options (tailored services etc.) in the `configure` script call – all options must be passed at once.

Listing 10 presents complete configuration of cross-build compilation for SAMV71, SAMRH71 and SAMRH707 ARM platforms. Listing 11 presents the configuration for GR712RC LEON3 platform. Those configurations were used during Library validation activities.

Listing 10 – Cross-compilation build configuration example (ARM platforms).

```
$ ./configure --host=arm-none-eabi \
       "LDFLAGS= -mcpu=cortex-m7 \
                 -mfloat-abi=hard \
                 -mfpu=fpv5-d16 \
                 -mlittle-endian \
                 -mthumb -ffunction-sections \
                 -Wl,--gc-sections \
                 --specs=nosys.specs" \
       "CFLAGS=  -O2 \
                 -ggdb3 \
                 -DCO_DEFTYPE_TIME_SCET=0x0060 \
                 -DCO_DEFTYPE_TIME_SUTC=0x0061 \
                 -DLELY_HAVE_ITIMERSPEC=1 \
                 -mcpu=cortex-m7 \
                 -mfloat-abi=hard \
                 -mfpu=fpv5-d16 \
                 -mlittle-endian \
                 -mthumb \
                 -ffunction-sections" \
       --enable-ecss-compliance \
       --disable-shared \
       --disable-python \
       --disable-tests \
       --disable-unit-tests \
       --disable-threads
```


ESA Contract No. 4000143237/23/NL/AS

Listing 11 – Cross-compilation build configuration example (GR712RC platform).

```
$ ./configure --host=sparc-gaisler-elf \
       "LDFLAGS= -qbsp=gr712rc \
                 -mcpu=leon3 \
                 -mfix-gr712rc \
                 -Wl,--gc-sections" \
       "CFLAGS=  -O2 \
                 -ggdb3 \
                 -DCO_DEFTYPE_TIME_SCET=0x0060 \
                 -DCO_DEFTYPE_TIME_SUTC=0x0061 \
                 -DLELY_HAVE_ITIMERSPEC=1 \
                 -DLELY_HAVE_TIMESPEC=1 \
                 -DLELY_HAVE_SYS_TYPES_H=1 \
                 -mcpu=leon3 \
                 -qbsp=gr712rc \
                 -mfix-gr712rc" \
       --enable-ecss-compliance \
       --disable-shared \
       --disable-python \
       --disable-tests \
       --disable-unit-tests \
       --disable-threads
```

11.2.3 Installation

Assuming that the Library is already built, it is enough to execute `make install` to install the library at default system's location. In order to control the target installation directory, one can set it up at the configure level (Listing 12).

Listing 12 – Custom installation directory configuration.

```
$ ./configure --disable-shared --enable-ecss-compliance --prefix /custom/canopen/directory
```

11.2.4 Include paths and library dependencies

The installed Library has a very simple structure:

```
include/
lib/
share/
```

The `lib/pkgconfig/` subdirectory contains `pkg-config` metadata files and in order to use Library, the user application should use the `liblely-co.pc` file for configuration.

In case `pkg-config` is not used by the user and the library is not installed in the system's default location, `include/` directory should be added to include paths e.g.

```
$ gcc <other options> -isystem /path/to/library/include
```

and `lib/` directory added to lib path with `-llely-compat`, `-llely-util`, `-llely-can` and `-llely-co` libraries (in that order) linked into the user application executable file.

ESA Contract No. 4000143237/23/NL/AS

11.2.5 Endianness considerations

CANopen library stores its Object Dictionary always as Little Endian, disregarding platform memory layout. This makes the Object Dictionary contents consistent with CANopen messages endianness, which is useful with services like PDO and SDO. This puts additional responsibilities on the user when writing portable code: user should never modify Object Dictionary contents directly (via pointer etc.). While accessing Object Dictionary user should always rely on library access functions (from `co_dev_get_val_X_` and `co_dev_set_val_X_` families – for example `co_dev_get_val_u32`). Do not use "typeless" `co_dev_get_val` or `co_dev_set_val` for value types other than DOMAIN.

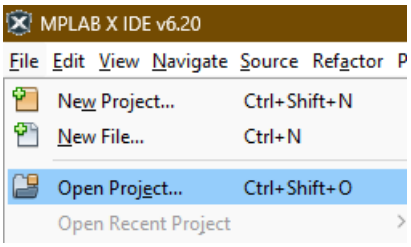
11.2.6 Using CANSW with Microchip MPLAB X IDE

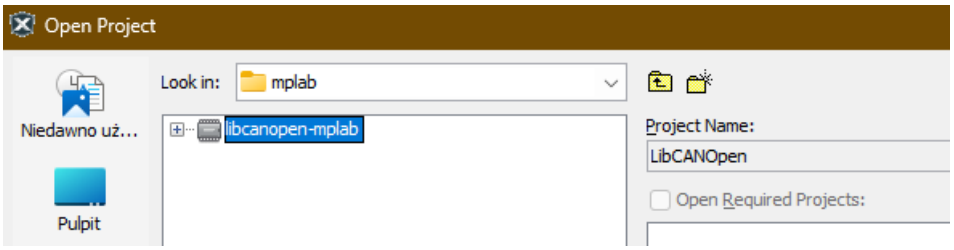
CANopen library can be integrated with Microchip's MPLAB IDE using Python script provided as a part of CTESW.

The script generates a library project containing CANopen library sources. That project can be easily added to MPLAB project as an external library, to be used in CANopen applications. Details about the script's usage can be found in CTESW SUM [RD8].

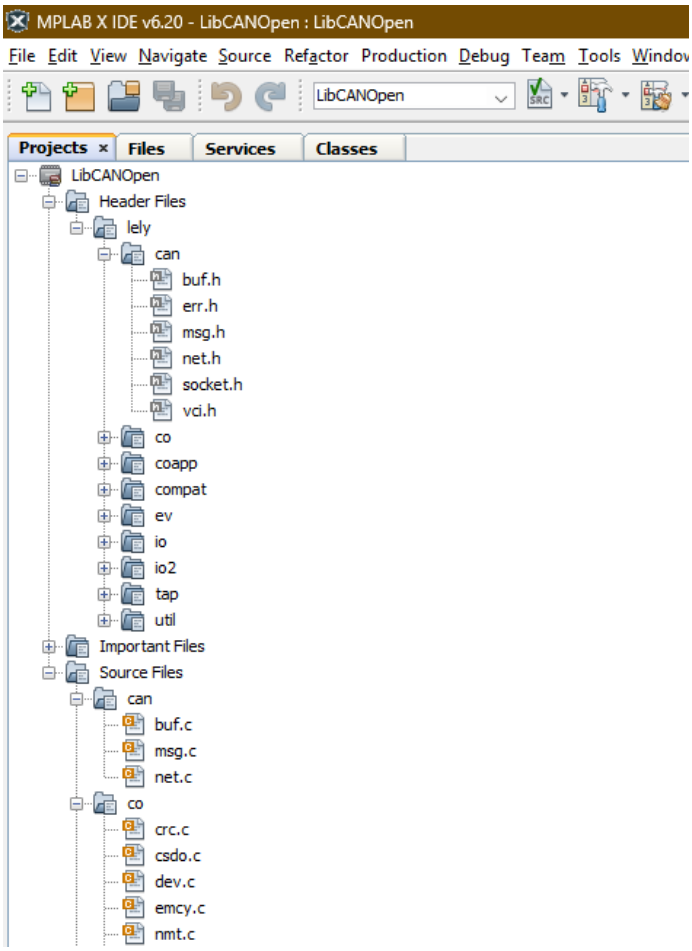
Source code distribution also contains pre-prepared MPLAB IDE packages.

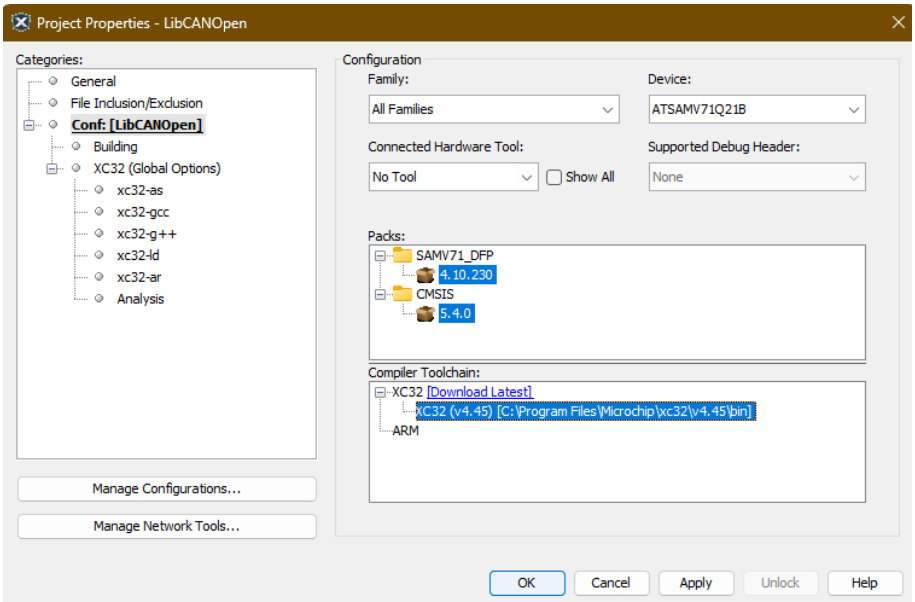
After obtaining the library in MPLAB format, the project can be opened in MPLAB X IDE:





After opening the project, it's recommended to verify that all the files are present, and correct toolchain is used for building the project, as it's not explicitly specified by the script.





After selecting the compiler, the project can be built and included in other MPLAB X IDE projects.

It's important to note that the include directory for LibCANopen must be manually specified in application project, and should point to `include/` subdirectory of generated MPLAB library project.

11.3 Using the software on a typical task

11.3.1 Memory allocation

Many operations performed by the Library require allocation of some amount of memory when needed. The Library contains a unified interface for handling memory allocation and deallocation. That interface provides an opt-in method for hooking up any custom memory allocation scheme that might serve project or application specific requirements.

The API for memory allocation is provided by the `lely/util/memory.h` header file. The most important functions declared there are `mem_alloc` and `mem_free` responsible for allocating and freeing memory, respectively. Both serve as the public interface for the chosen underlying allocator, that has to be passed as an argument.

The generic interface is realized by the `alloc_vtbl` structure that's mostly referenced through its `alloc_t` type alias (see Listing 13).

Listing 13 – `alloc_t` allocator structure.

```
struct alloc_vtbl {
        void *(*alloc)(alloc_t *alloc, size_t alignment, size_t size);
        void (*free)(alloc_t *alloc, void *ptr);
        size_t (*size)(const alloc_t *alloc);
        size_t (*capacity)(const alloc_t *alloc);
};
```

It is the user's responsibility to provide all four functions that match the signatures of above mentioned function pointers with the implementation of an application-specific allocator. In a later chapter we'll cover an example implementation of such allocator.

Inside the Library, all references to the selected allocator are made through the `can_net_t` structure. Therefore the user has to provide a pointer to the allocator when creating the CAN Network object as shown in Listing 14.

Listing 14 – `alloc_t` and `can_net_t` setup.

```
alloc_t* allocator = create_allocator();  // user-provided
can_net_t* network = can_net_create(allocator);
```

It is also possible to use the allocator manually, outside the Library code.

11.3.1.1 Default allocator

If no allocator is provided then the default one will be used inside the Library.

Listing 15 – Default (NULL) allocator.

```
void* allocated_memory = mem_alloc(NULL);
...
mem_free(NULL, allocated_memory);
```

The ECSS-compliant compilation of the Library provides an empty implementation of the default memory allocator. In other compilation modes it is based on dynamic heap memory and is thus of very

ESA Contract No. 4000143237/23/NL/AS

little use for resource-constrained projects. But it's worth noting that such behaviour of passing a null pointer is well-defined (Listing 15).

The empty implementation available in ECSS-compliant compilation does not perform any allocation, which makes most of the Library features unavailable. Hence the user is responsible for providing allocator, either a custom one or properly configured pool allocator (11.3.1.2).

11.3.1.2 Custom pool allocator

A very likely popular use case is to use an arena-style allocator. The Library provides a utility structure `mempool` that implements a very basic memory pool allocator. Listing 16 shows example use scheme.

Listing 16 – Pool allocator usage example.

```
#include <lely/co/type.h>
#include <lely/util/memory.h>
#include <lely/util/mempool.h>
// ...
alloc_t* create_allocator() {
  const size_t POOL_SIZE = 128u * 1024u;
  static co_unsigned8_t memory[POOL_SIZE] = {0};
  static mempool pool;

  return mempool_init(&pool, memory, POOL_SIZE);
}
```

Total allocated size and remaining pool capacity can be then queried using `mem_size` and `mem_capacity`.

Above presented memory allocator is a very simple one. It uses 128kB of static memory in the executable to provide storage for further on-demand allocation. It does not actually mark deallocated memory as available and is thus unable to reuse it later. The Library itself doesn't create short-lived objects in ECSS-compliant compilation, but a more complex memory allocation strategy might be preferred by users, depending on their project needs.

11.3.2 Receiving and sending CAN frames

The Library provides an implementation of the CANopen protocol, which is a Layer 3 and above network protocol, according to the OSI model. Below layers are not implemented, must be available separately and their integration with the Library is expected to be provided by the user.

The Library uses a generic structure to represent CAN frames or messages. It consists of a 32-bit unsigned integer to represent a CAN-ID (either 11 or 29 bit long), a byte for bit flags (e.g. to denote an RTR frame), a byte to store the number of bytes in the data field and the data field itself (Listing 17).

Listing 17 – `can_msg` structure representing CAN frame.

```
struct can_msg {
  uint_least32_t id;
  uint_least8_t flags;
  uint_least8_t len;
  uint_least8_t data[CAN_MSG_MAX_LEN];
};
```

With `CAN_MSG_MAX_LEN` being equal to either 8 or 64 depending on whether CAN FD support is enabled. In ECSS compliance mode it is disabled and CAN frames can store up to 8 bytes of data.

It is the user's responsibility to create proper instances of the above explained data structure and to pass them to the Library. Conversely, it's also the responsibility of the user to translate any instance provided by the Library to the proper target representation that could be encoded on the wire.

In order to inform the Library that a new CAN frame was received and should be handled, one should use the `can_net_recv` function as shown in Listing 18.

Listing 18 – CAN frame receiving using `can_net_recv` usage example.

```
#include <lely/can/msg.h>
#include <lely/can/net.h>
// ...
alloc_t* allocator = create_allocator();  // user-provided
can_net_t* network = can_net_create(allocator);
// ...
struct can_msg msg = prepare_message();  // user-provided
const int return_code = can_net_recv(network, &msg, 0);
```

In order to obtain a CAN frame to be sent from the Library, one should provide a callback function that will be called with the frame every time the Library requests a frame to be sent. That function must conform to the `int (const struct can_msg *msg, void *data)` signature and can be installed using `can_net_set_send_func`. Listing 19 presents an example of callback setup.

Listing 19 – CAN frame sending callback example.

```
// user-provided
int send_func(const can_msg* msg, int bus_id, void* data) {
  printf("SEND <%#x> <%#x>", msg->id, msg->flags);
  return 0;
}

void init_network() {
  alloc_t* allocator = create_allocator();  // user-provided
  can_net_t* network = can_net_create(allocator);
  can_net_set_send_func(network, &send_func, NULL);
// ...
```

Presented code only prints on the standard output some details of the frame that the Library requested to be sent. A production-level implementation should interpret all fields of `msg` and attempt to send them on the actual CAN bus.

ESA Contract No. 4000143237/23/NL/AS

11.3.3 SocketCAN frame translation

The former chapter explained the need for translating between a generic CAN frame format structure provided by the Library and the target network stack. Choice of the latter is project-dependent. The purpose of this chapter is to provide a concrete example using SocketCAN. SocketCAN is the default driver for CAN networking stack in the Linux Kernel, therefore it should be quite popular. Additionally the translation itself is pretty straightforward. This makes it a good candidate to use as an example.

Assuming that there is already an instance of `struct can_frame` read from a SocketCAN socket, one can execute the code from Listing 20 to produce a `struct can_msg` that can be then provided to `can_net_recv` later.

Listing 20 – Example of `can_msg` creation from lower layer data (SocketCAN).

```
int convert_canframe_to_canmsg(const struct can_frame* from,
                               struct can_msg* to) {
  if (from->can_id & CAN_ERR_FLAG) {
    return -1;
  }

  *to = CAN_MSG_INIT;

  if (from->can_id & CAN_EFF_FLAG) {
    to->id = from->can_id & CAN_EFF_MASK;
    to->flags |= CAN_FLAG_IDE;
  } else {
    to->id = from->can_id & CAN_SFF_MASK;
  }

  if (from->can_id & CAN_RTR_FLAG) {
    to->flags |= CAN_FLAG_RTR;
  }

  to->len = MIN(from->can_dlc, CAN_MAX_LEN);

  if (!(to->flags & CAN_FLAG_RTR)) {
    memcpy(to->data, from->data, to->len);
  }

  return 0;
}
```

Translation in the other direction is equally straightforward. It's enough to reverse the performed operations as shown in Listing 21.

ESA Contract No. 4000143237/23/NL/AS

Listing 21 – Example of `can_msg` transformation to lower layer data (SocketCAN).

```
void convert_canmsg_to_canframe(const struct can_msg* from,
                                struct can_frame* to) {
  memset(to, 0, sizeof(*to));

  to->can_id = from->id;

  if (from->flags & CAN_FLAG_IDE) {
    to->can_id &= CAN_EFF_MASK;
    to->can_id |= CAN_EFF_FLAG;
  } else {
    to->can_id &= CAN_SFF_MASK;
  }

  to->can_dlc = MIN(from->len, CAN_MAX_LEN);

  if (from->flags & CAN_FLAG_RTR) {
    to->can_id |= CAN_RTR_FLAG;
  } else {
    memcpy(to->data, from->data, to->can_dlc);
  }
}
```

After calling the above function, one has an instance of `struct can_frame` that is ready to be written to the SocketCAN socket.

11.3.4 Setting the time / external clock

The Library does not assume a specific clock on a target platform. It is entirely the responsibility of the user to provide current time information to the Library. As with the chosen memory allocator, current time is a property of the CAN network interface. One can set it using `can_net_set_time`. The function accepts a value of standard `timespec` type, and if unavailable the `<lely/compat/time.h>` header file provides a compatible definition. It's a simple structure of two values `tv_sec` and `tv_nsec` to represent a time point using number of seconds and nanoseconds in between them. The time provided should come from a monotonic clock i.e. the provided value should never be smaller than the one provided before.

Listing 22 – CAN network time setting using `can_net_set_time` example.

```
#include <time.h>
// ...
void update_time() {
  struct timespec ts = {0};
  clock_gettime(CLOCK_MONOTONIC, &ts);

  const int rc = can_net_set_time(&ts);
  // user-provided code to handle a non-zero value of rc
}
```

Calling `can_net_set_time` function (example shown in Listing 22) may invoke multiple registered timer callback functions. For example it might trigger sending a CANopen SYNC message if there is a

ESA Contract No. 4000143237/23/NL/AS

SYNC producer service running. One of those callback functions may fail, that's the reason the `can_net_set_time` function provides a return code.

The time point at which the next timer callback function will be triggered can be queried. The CAN network interface provides another callback that can be set using the `can_net_set_next_func` function, shown in example on Listing 23. This way user code can schedule time updates for proper moments, to achieve the best timer events precision.

Listing 23 – CAN network callback for re-scheduling time update.

```
int next_func(const timespec* ts, void* data) {
  printf("Next trigger at: %ld.%ld\n", ts->tv_sec, ts->tv_nsec);
  return 0;
}
// ...
can_net_set_next_func(network, &next_func, NULL);
// ...
update_time();  // user-provided, calls can_net_set_time
```

11.3.5 Device and Object Dictionary

Two most central concepts in the CANopen protocol are the CANopen Device and the Object Dictionary that's part of the former. Typically in a single application there will be one instance of the device structure that contains its own dictionary. The dictionary is a set of objects that can be accessed using a 16-bit index. Each object can represent a singular value, an array or a record of up to 255 smaller sub-object values. Part of the dictionary is used to configure the CANopen device communication parameters (index range between 0x1000 and 0x1fff) and part of it is available for storing application specific data.

Usually the dictionary is initialized based on the contents of EDS (Electronic Data Sheet) or DCF (Device Configuration File) files. This tutorial shows how to create such entries manually using the provided API.

The device, object and sub-object instances can be allocated using a custom memory allocator, like the one presented in a former chapter. But for the sake of simplicity, static memory will be used to provide storage for those instances.

First one needs to create and initialize a device. Code in Listing 24 does that using a Node-ID of 0x01.

Listing 24 – CANopen device (`co_dev_t`) initialization example.

```
#include <lely/co/detail/dev.h>

static co_dev_t device;

int main() {
  co_dev_init(&device, 0x01u);
}
```

With the device ready, it's time to create an entry in its object dictionary – see Listing 25.

Listing 25 – CANopen Object Dictionary entry creation example.

```
#include <lely/co/detail/obj.h>

// ...

static struct {
  co_unsigned32_t sub0;
} value1005 = {
  .sub0 = 0x40000080;
};

static co_obj_t object1005;
static co_sub_t object1005sub0;

int main() {
  // ...

  co_obj_init(&object1005, 0x1005u, &value1005, sizeof(value1005));
  co_sub_init(&object1005sub0, 0x00u, CO_DEFTYPE_UNSIGNED32,
              &value1005.sub0);

  co_dev_insert_obj(&device, &object1005);
  co_obj_insert_sub(&object1005, &object1005sub0);
}
```

The snippet from Listing 25 creates an object dictionary part that represents the `COB-ID SYNC message` object with the default CAN-ID value (0x80) and the gen. bit set. The `value1005` structure serves as the storage for object and sub-object values, the `object1005` and `object1005sub0` provide storage for object metadata like index or data type.

11.3.6 dcf2dev

Manually specifying entire contents of the Object Dictionary is a laborious and error-prone process. To alleviate that, CANSW includes a Python-based tool `dcf2dev`, that can optionally be used to transform human-readable Device Configuration Files (`DCF`) into C code that initializes a device with its object dictionary fully populated. It's located in the `python/dcf_tools` directory. In order to make use of it, it has to be installed.

A standard `setuptools` installation process is supported as shown in Listing 26.

Listing 26 – Standard `dcf2dev` installation using `setuptools`.

```
$ cd python/dcf-tools
$ python setup.py install
```

It is highly recommended to perform above in a virtual environment, so the tool will not collide with other Python modules preinstalled in the user's system. For example one can use module `venv` available in Python 3.3 and newer, as shown in Listing 27.

Listing 27 – `dcf2dev` installation in virtual environment (using `venv`).

```
$ cd ~/<user-folder-to-store-venv-files>
$ python3 -m venv my_venv
$ source my_venv/bin/activate
$ cd <path-to-CANSW>/python/dcf_tools
$ python3 setup.py install
```

Successful installation of dcf2dev can be verified by invoking the tool (see Listing 28), which should return its usage instructions.

Listing 28 – `dcf2dev` help command.

```
$ dcf2dev --help
```

After installation the tool can be used to transform a DCF file into a pair of source and header files. It accepts a name of the input file and a module name to use in generated code – see Listing 29.

Listing 29 – `dcf2dev` usage.

```
$ dcf2dev [--header] [--include-config] file_name.dcf module_name
```

Results are printed on standard output, one may redirect them straight to a regular file as in Listing 30.

Listing 30 – `dcf2dev` usage example.

```
$ dcf2dev --header tutorial.dcf tutorial > tutorial.h
$ dcf2dev --include-config tutorial.dcf tutorial > tutorial.c
```

If user's application uses a build system to pass configuration macros using compiler flags instead of the `config.h` header file, then the option `--include-config` should not be used.

For example, after using the tool, following the exact commands mentioned in Listing 30, on a minimal DCF from Listing 31, it will generate a C header file and a matching C code source file, shown in Listing 32 and Listing 33 respectively.

Listing 31 – Minimal DCF file example.

```
[DeviceInfo]
VendorName=
VendorNumber=0
ProductName=
ProductNumber=0
RevisionNumber=0
OrderCode=
BaudRate_10=0
BaudRate_20=0
BaudRate_50=0
BaudRate_125=0
BaudRate_250=0
BaudRate_500=0
BaudRate_800=0
BaudRate_1000=0

[MandatoryObjects]
SupportedObjects=3
1=0x1000
```


ESA Contract No. 4000143237/23/NL/AS

```
2=0x1001
3=0x1018

[OptionalObjects]
SupportedObjects=0

[ManufacturerObjects]
SupportedObjects=0

[1000]
ParameterName=Device type
DataType=0x0007
AccessType=ro

[1001]
ParameterName=Error register
DataType=0x0005
AccessType=ro

[1018]
SubNumber=5
ParameterName=Identity object
ObjectType=0x09

[1018sub0]
ParameterName=Highest sub-index supported
DataType=0x0005
AccessType=const
DefaultValue=0x4

[1018sub1]
ParameterName=Vendor-ID
DataType=0x0007
AccessType=ro

[1018sub2]
ParameterName=Product code
DataType=0x0007
AccessType=ro

[1018sub3]
ParameterName=Revision number
DataType=0x0007
AccessType=ro

[1018sub4]
ParameterName=Serial number
DataType=0x0007
AccessType=ro
```


ESA Contract No. 4000143237/23/NL/AS

Listing 32 – Example of header file generated by `dcf2dev`.

```
#ifndef TUTORIAL_H_GENERATED_
#define TUTORIAL_H_GENERATED_

#if !LELY_NO_MALLOC
#error Static object dictionaries are only supported when dynamic memory
allocation is disabled.
#endif

#include <lely/co/co.h>

#ifdef __cplusplus
extern "C" {
#endif

  co_dev_t * tutorial_init(void);

#ifdef __cplusplus
}
#endif

#endif // TUTORIAL_H_GENERATED_
```

Listing 33 – Excerpts from example C source file generated by `dcf2dev`.

```
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#if !LELY_NO_MALLOC
#error Static object dictionaries are only supported when dynamic memory
allocation is disabled.
#endif

#include <lely/co/detail/dev.h>
#include <lely/co/detail/obj.h>
#include <lely/util/cmp.h>

// ...
// static definitions of all required data structures and their instances
// ...

co_dev_t *
tutorial_init(void) {
  static co_dev_t *dev = NULL;
  if (!dev) {
    dev = &tutorial;

    co_dev_insert_obj(&tutorial, &tutorial_1000);
    co_obj_insert_sub(&tutorial_1000, &tutorial_1000sub0);

    co_dev_insert_obj(&tutorial, &tutorial_1001);
```


ESA Contract No. 4000143237/23/NL/AS

```
    co_obj_insert_sub(&tutorial_1001, &tutorial_1001sub0);

    co_dev_insert_obj(&tutorial, &tutorial_1018);
    co_obj_insert_sub(&tutorial_1018, &tutorial_1018sub0);
    co_obj_insert_sub(&tutorial_1018, &tutorial_1018sub1);
    co_obj_insert_sub(&tutorial_1018, &tutorial_1018sub2);
    co_obj_insert_sub(&tutorial_1018, &tutorial_1018sub3);
    co_obj_insert_sub(&tutorial_1018, &tutorial_1018sub4);
  }
  return dev;
}
```

User code should `#include` the generated `tutorial.h` file and call function `tutorial_init()` to obtain a ready to use device instance. The `tutorial.c` file should be separately compiled and its generated object code linked into the application.

Even though the generated source code is in the C Programming Language and must be compiled as such, the generated header file is prepared to also be consumed by application code written in C++.

11.3.7 NMT Master and Slave

Based on the foundations already established by preceding tutorial chapters, it's time to create a working example with two nodes based on CANSW exchanging data with each other. It will involve an application serving as an NMT Master node and the other one filling the role of NMT Slave. The former will produce and send SYNC messages and receive NMT Heartbeat messages. The latter will have a reverse responsibility, it will consume SYNC messages and produce heartbeats. This chapter serves as a basis for subsequent chapters, which cover incremental additions to more CANopen-based tasks.

First we will start with minimalistic DCF files for both Master (Listing 34) and Slave (Listing 35) applications.

Listing 34 – Minimal NMT Master DCF example.

```
[DeviceInfo]
VendorName=European Space Agency
VendorNumber=0x0000033E
BaudRate_10=1
BaudRate_20=1
BaudRate_50=1
BaudRate_125=1
BaudRate_250=1
BaudRate_500=1
BaudRate_800=1
BaudRate_1000=1

[DeviceComissioning]
NodeID=0x01

[MandatoryObjects]
SupportedObjects=3
1=0x1000
2=0x1001
3=0x1018
```


ESA Contract No. 4000143237/23/NL/AS

```
[OptionalObjects]
SupportedObjects=5
1=0x1005
2=0x1006
3=0x1016
4=0x1F80
5=0x1F81

[ManufacturerObjects]
SupportedObjects=0

[1000]
ParameterName=Device type
DataType=0x0007
AccessType=ro

[1001]
ParameterName=Error register
DataType=0x0005
AccessType=ro

[1005]
ParameterName=COB-ID SYNC message
DataType=0x0007
AccessType=rw
DefaultValue=0x40000080

[1006]
ParameterName=Communication cycle period
DataType=0x0007
AccessType=rw
DefaultValue=500000

[1016]
SubNumber=2
ParameterName=Consumer heartbeat time
ObjectType=0x09

[1016sub0]
ParameterName=Highest sub-index supported
DataType=0x0005
AccessType=const
DefaultValue=0x01

[1016sub1]
ParameterName=Consumer heartbeat time
DataType=0x0007
AccessType=rw
DefaultValue=0x00020226

[1018]
SubNumber=5
```


ESA Contract No. 4000143237/23/NL/AS

```
ParameterName=Identity object
ObjectType=0x09

[1018sub0]
ParameterName=Highest sub-index supported
DataType=0x0005
AccessType=const
DefaultValue=0x04

[1018sub1]
ParameterName=Vendor-ID
DataType=0x0007
AccessType=ro
DefaultValue=0x0000033E

[1018sub2]
ParameterName=Product code
DataType=0x0007
AccessType=ro

[1018sub3]
ParameterName=Revision number
DataType=0x0007
AccessType=ro

[1018sub4]
ParameterName=Serial number
DataType=0x0007
AccessType=ro

[1F80]
ParameterName=NMT startup
DataType=0x0007
AccessType=rw
ParameterValue=0x00000001

[1F81]
ParameterName=NMT slave assignment
ObjectType=0x08
DataType=0x0007
AccessType=rw
CompactSubObj=2

[1F81Value]
NrOfEntries=1
2=0x00000001
```

Listing 35 – Minimal NMT Slave DCF example.

```
[DeviceInfo]
VendorName=European Space Agency
VendorNumber=0x0000033E
BaudRate_10=1
```

```
BaudRate_20=1
BaudRate_50=1
BaudRate_125=1
BaudRate_250=1
BaudRate_500=1
BaudRate_800=1
BaudRate_1000=1

[DeviceComissioning]
NodeID=0x02

[MandatoryObjects]
SupportedObjects=3
1=0x1000
2=0x1001
3=0x1018

[OptionalObjects]
SupportedObjects=3
1=0x1005
2=0x1017
3=0x1F80

[ManufacturerObjects]
SupportedObjects=0

[1000]
ParameterName=Device type
DataType=0x0007
AccessType=ro

[1001]
ParameterName=Error register
DataType=0x0005
AccessType=ro

[1005]
ParameterName=COB-ID SYNC message
DataType=0x0007
AccessType=rw
DefaultValue=0x00000080

[1017]
ParameterName=Producer heartbeat time
DataType=0x0006
AccessType=rw
DefaultValue=500

[1018]
SubNumber=5
ParameterName=Identity object
ObjectType=0x09
```


ESA Contract No. 4000143237/23/NL/AS

```
[1018sub0]
ParameterName=Highest sub-index supported
DataType=0x0005
AccessType=const
DefaultValue=0x04

[1018sub1]
ParameterName=Vendor-ID
DataType=0x0007
AccessType=ro
DefaultValue=0x0000033E

[1018sub2]
ParameterName=Product code
DataType=0x0007
AccessType=ro

[1018sub3]
ParameterName=Revision number
DataType=0x0007
AccessType=ro

[1018sub4]
ParameterName=Serial number
DataType=0x0007
AccessType=ro

[1F80]
ParameterName=NMT startup
DataType=0x0007
AccessType=rw
DefaultValue=0x00000000
```

Above files should be used as input to `dcf2dev` to generate appropriate C code that can be compiled and linked into the Master and Slave applications, respectively. With that done, one can start implementing the basic application code structure, starting with necessary `#include` directives listed in Listing 36.

ESA Contract No. 4000143237/23/NL/AS

Listing 36 – Necessary `#include` directives for minimal NMT implementation.

```
#include "config.h"  // generated while configuring the Library

#include <signal.h>
#include <stdio.h>
#include <time.h>

#include <lely/can/msg.h>
#include <lely/can/net.h>

#include <lely/co/dev.h>
#include <lely/co/obj.h>
#include <lely/co/nmt.h>

#include <lely/util/error.h>
#include <lely/util/memory.h>
#include <lely/util/mempool.h>
```

Listing 37 shows 3 functions the implement the requirements of a hardware abstraction layer that needs to be provided by the user. One function is extracted with common code for observability purposes. Everything was explained in previous chapters, but the `receive_message` needs an additional comment. It is assumed in this tutorial that the user provided `receive_convert` function returns CAN frame from internal hardware buffer or `NULL` if no CAN frame was received by the underlying hardware.

ESA Contract No. 4000143237/23/NL/AS

Listing 37 – Hardware abstraction layer implementation example.

```c
void print_msg(const char* func, const struct can_msg* msg) {
  printf("%s <%#x> <%#x>", func, msg->id, msg->flags);

  if (msg->len != 0) printf(" DATA:");
  for (size_t i = 0; i < msg->len; ++i) printf(" 0x%02x", msg->data[i]);

  printf("\n");
}

int send_func(const struct can_msg* msg, int bus_id, void* data) {
  print_msg("SEND", msg);
  return convert_and_send(msg);  // user-provided
}

void receive_message(can_net_t* network) {
  struct can_msg* msg = receive_convert();  // user-provided
  if (msg == NULL)
    return;
  print_msg("RECV", msg);
  can_net_recv(network, msg);
}

void set_current_time(can_net_t* network) {
  struct timespec ts;
  get_current_time(&ts);  // user-provided
  can_net_set_time(network, &ts);
}
```

With a very barebone HAL described above, an example framework for both applications can be introduced – see Listing 38. The only difference between Master and Slave applications is in the use of `dcf2dev` generated code. After a small setup, the applications are run until the `SIGINT` signal is received. The main loop checks for new data on CAN bus, reads new frame if available, and updates the clock. This basic loop is a good fit for tutorial purposes but production grade programs will likely use a more complex event-loop based solution.

The final code block in the listing is responsible for creating and starting an NMT service instance. Additionally two indication functions are set for observability purposes – see Listing 39.

After building both applications and running them for a while, it should be possible to observe continuous data exchange between them. Example output can be found in Listing 40. It can be seen in it that an NMT boot-up was sent and later an NMT "reset communication" command was sent. Heartbeat and state indication functions are called. SYNC message (CAN-ID 0x80) is being sent and Heartbeat message originated at the slave node is received.

Listing 38 – Basic example of NMT service framework for Master/Slave application.

```
// necessary #include directives, see Listing 36
#include "dcf_nmt.h"  // from dcf2dev

static volatile sig_atomic_t last_signal;
void signal_handler(int sig) { last_signal = sig; }
// system integration functions, see Listing 37
// NMT Service indication functions, see Listing 39

co_nmt_t* setup_nmt(co_dev_t* device, can_net_t* network) {
  co_nmt_t* nmt = co_nmt_create(network, device);
  assert(nmt != NULL);

  co_nmt_set_hb_ind(nmt, &hb_ind, NULL);
  co_nmt_set_st_ind(nmt, &st_ind, NULL);

  co_nmt_cs_ind(nmt, CO_NMT_CS_RESET_NODE);
}

int main() {
  signal(SIGINT, &signal_handler);

  co_dev_t* device = dcf_nmt_init();  // from dcf2dev
  alloc_t* alloc = create_allocator();
  can_net_t* network = can_net_create(alloc);
  can_net_set_send_func(network, &send_func, NULL);

  co_nmt_t* nmt = setup_nmt(device, network);

  set_current_time(network);

  while (last_signal == 0) {
    receive_message(network);
    set_current_time(network);
  }

  co_nmt_destroy(nmt);
  can_net_destroy(network);

  return 0;
}
```

Listing 39 – Basic NMT service indication functions example.

```
void hb_ind(co_nmt_t* nmt, co_unsigned8_t id, int state, int reason,
            void* data) {
  printf("HB: id=<%d> state=<%d> reason=<%d>\n", id, state, reason);
}

void st_ind(co_nmt_t* nmt, co_unsigned8_t id, co_unsigned8_t st,
            void* data) {
  printf("ST: id=<%d> st=<%d>\n", id, st);
}
```


ESA Contract No. 4000143237/23/NL/AS

Listing 40 – Sample output from the NMT master node application.

```
ST: id=<1> st=<0>
ST: id=<1> st=<0>
SEND <0x701> <0> DATA: 0x00
ST: id=<1> st=<127>
SEND <0> <0> DATA: 0x82 0x00
RECV <0x702> <0> DATA: 0x7f
HB: id=<2> state=<0> reason=<1>
ST: id=<2> st=<127>
RECV <0x702> <0> DATA: 0x00
HB: id=<2> state=<0> reason=<1>
ST: id=<2> st=<0>
ST: id=<2> st=<0>
SEND <0x80> <0>
SEND <0x80> <0>
RECV <0x702> <0> DATA: 0x7f
```

11.3.8 PDO

This chapter shows how to configure a simple PDO-based data transmission between two CANopen nodes. Based on the foundation built in former chapter, a manufacturer-specific object is added to both master and slave node applications. Master uses 0x3000 as its index and has it mapped to a synchronous cyclic TPDO. Slave on the other hand uses 0x4000 for its object, and has it mapped to a synchronous RPDO that matches the master's TPDO. The master application built in former chapter produces a SYNC message and newly introduced PDO-based data transmission is synchronized to both sending and reception of that message. Listing 41 presents necessary modifications of the Master's DCF file from previous chapter to add simple PDO service.

Listing 41 – NMT Master DCF file modifications for example PDO service.

```
# ...
[OptionalObjects]
SupportedObjects=7
1=0x1005
2=0x1006
3=0x1016
4=0x1F80
5=0x1F81
6=0x1800
7=0x1A00

[ManufacturerObjects]
SupportedObjects=1
1=0x3000
# ...
[3000]
ParameterName=TutorialMasterTime
DataType=0x0007  # 32-bit unsigned integer
AccessType=rw
DefaultValue=0
PDOMapping=1
```

```
[1800]
SubNumber=3
ParameterName=TPDO communication parameter
ObjectType=0x09

[1800sub0]
ParameterName=Highest sub-index supported
DataType=0x0005
AccessType=const
DefaultValue=0x02

[1800sub1]
ParameterName=COB-ID used by TPDO
DataType=0x0007
AccessType=rw
DefaultValue=$NODEID+0x180

[1800sub2]
ParameterName=Transmission type
DataType=0x0005
AccessType=rw
DefaultValue=0x01  # cyclic every sync

[1A00]
ParameterName=TPDO mapping parameter
ObjectType=0x09
DataType=0x0007
AccessType=rw
CompactSubObj=1

[1A00Value]
NrOfEntries=1
1=0x30000020
```

A basic manufacturer-specific object 0x3000 representing a 32-bit unsigned integer is added with PDO mapping enabled. To configure the TPDO service, objects 0x1800 and 0x1A00 are also added. The 0x3000 object is mapped in 0x1A00. The COB-ID used by object 0x1800 is equal to 0x181, the RPDO service on the slave node will use the same value.

After regenerating the device initialization code using `dcf2dev`, only a small modification is necessary to master application code as presented in Listing 42.

Listing 42 – Master application modifications for example PDO service.

```
void update_time_object(co_dev_t* device) {
  co_obj_t* obj = co_dev_find_obj(device, 0x3000u);
  if (!obj)
    return;

  const co_unsigned32_t value = time(NULL);

  co_obj_set_val(obj, 0x00u, &value, sizeof(value));
}
// ...
// main loop
while (last_signal == 0) {
  receive_message(network);
  update_time_object(device);  // newly added line
  set_current_time(network);
 }
```

Above code snippet looks for the 0x3000 object and stores the current UNIX timestamp value in it on every loop iteration. This is the information that will be later read by the slave application.

After building and running the application, the output from Listing 43 should be visible. TPDO is transmitted right after sending a SYNC message and it can be seen that the value sent is slowly being incremented by one. SYNC is configured to be sent every 500 milliseconds, which explains why the same timestamp value is sent twice.

Listing 43 – Output from running example Master application with TPDO service.

```
SEND <0x80> <0>
SEND <0x181> <0> DATA: 0xc7 0x7a 0xd8 0x60
SEND <0x80> <0>
SEND <0x181> <0> DATA: 0xc8 0x7a 0xd8 0x60
SEND <0x80> <0>
SEND <0x181> <0> DATA: 0xc8 0x7a 0xd8 0x60
SEND <0x80> <0>
SEND <0x181> <0> DATA: 0xc9 0x7a 0xd8 0x60
```

It's time to move to the application running on the Slave node. Listing 44 presents the changes that need to be made to its DCF file.

ESA Contract No. 4000143237/23/NL/AS

Listing 44 – NMT Slave DCF file modifications for example PDO service.

```
# ...
[OptionalObjects]
SupportedObjects=5
1=0x1005
2=0x1017
3=0x1F80
4=0x1400
5=0x1600

[ManufacturerObjects]
SupportedObjects=1
1=0x4000
# ...
[4000]
ParameterName=TutorialMasterTime
DataType=0x0007  # 32-bit unsigned integer
AccessType=rw
DefaultValue=0
PDOMapping=1

[1400]
SubNumber=3
ParameterName=RPDO communication parameter
ObjectType=0x09

[1400sub0]
ParameterName=Highest sub-index supported
DataType=0x0005
AccessType=const
DefaultValue=0x02

[1400sub1]
ParameterName=COB-ID used by RPDO
DataType=0x0007
AccessType=rw
DefaultValue=0x181

[1400sub2]
ParameterName=Transmission type
DataType=0x0005
AccessType=rw
DefaultValue=0x00

[1600]
ParameterName=RPDO mapping parameter
ObjectType=0x09
DataType=0x0007
AccessType=rw
CompactSubObj=2

[1600Value]
```


ESA Contract No. 4000143237/23/NL/AS

```
NrOfEntries=1
1=0x40000020
```

As with the Master application, a basic manufacturer-specific object representing a 32-bit unsigned integer is added with PDO mapping enabled. The only difference is that a different object index is used. To configure the RPDO service, objects 0x1400 and 0x1600 are also added. The 0x4000 object is then mapped in 0x1600. The COB-ID used by object 0x1400 is the same as the one used by Master's TPDO service.

Changes necessary for application code on the slave node are also quite small – see Listing 45. A custom download indication function for the 0x4000 object is set. That allows the application code to be notified on any writes made to the object. The tutorial uses it to simply print the received value. The `co_sub_dn` function performs the actual write of the received value into the object.

Listing 45 – Slave application modifications for example PDO service.

```
co_unsigned32_t tutorial_sub_dn_ind(co_sub_t* sub, struct co_sdo_req* req,
                                    co_unsigned32_t ac, void* data) {
  if (ac) return ac;

  co_sub_on_dn(sub, req, &ac);

  const co_unsigned32_t time_on_master = co_sub_get_val_u32(sub);
  printf("Time on master is %u\n", time_on_master);

  return ac;
}

void set_4000_dn_ind(co_dev_t* device) {
  co_obj_t* obj = co_dev_find_obj(device, 0x4000u);
  if (obj)
    co_obj_set_dn_ind(obj, &tutorial_sub_dn_ind, NULL);
}

// ...

co_nmt_t* nmt = setup_nmt(device, network);
set_4000_dn_ind(device);  // new line
set_current_time(network);
```

Assuming that the master application is still running, the slave application should now output messages similar to the ones in Listing 46.

Listing 46 – Output from running example Slave application with RPDO service.

```
RECV <0x181> <0> DATA: 0xd9 0x7c 0xd8 0x60
RECV <0x80> <0>
Time on master is 1624800473
RECV <0x181> <0> DATA: 0xda 0x7c 0xd8 0x60
SEND <0x702> <0> DATA: 0x05
RECV <0x80> <0>
Time on master is 1624800474
```

It can be seen that an RPDO is received, but it is only acted upon after receiving a SYNC message. That's when the custom indication function presented above is called and the received value is stored.

11.3.9 Client SDO

Because the 0x3000 object on the Master application is small and it takes only 4 bytes to represent its value, an expedited SDO transfer can also be used to obtain it upon request. There is a simple modification needed on the Slave application to enable a Client SDO Service, the 0x1280 object needs to be configured – see Listing 47.

Listing 47 – NMT Slave DCF file modifications for example Client SDO service.

```
# ...
[OptionalObjects]
SupportedObjects=6
# ...
6=0x1280
# ...

[1280]
SubNumber=4
ParameterName=SDO client parameter
ObjectType=0x09

[1280sub0]
ParameterName=Highest sub-index supported
DataType=0x0005
AccessType=const
DefaultValue=0x03

[1280sub1]
ParameterName=COB-ID client -> server (tx)
DataType=0x0007
AccessType=rw
DefaultValue=0x601

[1280sub2]
ParameterName=COB-ID server -> client (rx)
DataType=0x0007
AccessType=rw
DefaultValue=0x581

[1280sub3]
ParameterName=Node-ID of the SDO server
DataType=0x0005
AccessType=rw
DefaultValue=0x01
```

Given the above, one needs to access the CSDO service instance from the NMT instance and submit an upload request with the object index on the Master node, as shown in example method in Listing 48. Server SDO instance is already running on the Master node. The request can be submitted at any time the CSDO service is idle.

Listing 48 – Example code performing SDO request using Client SDO in Slave application.

```
void send_csdo_up_req(co_nmt_t* nmt) {
  co_csdo_t* csdo = co_nmt_get_csdo(nmt, 1);
  assert(csdo != NULL);

  const int rc = co_csdo_up_req(csdo, 0x3000, 0x00, NULL, &csdo_up_con, NULL);
  if (rc != 0) {
    printf("!! Failed to submit an SDO upload request due to error: %u\n",
           get_errnum());
  }
}
```

`csdo_up_con` is the user-provided upload confirmation function that will be called once the submitted request is finished, either successfully or not. For this tutorial definition from Listing 49 is used. First the abort code is checked. If it has a non-zero value, it means there was a failure and it should be inspected. Otherwise there is an attempt to read the value. We know that the 0x3000 object on the Master node is a 32-bit unsigned integer so exactly 4 bytes of data representing a UNIX timestamp are expected.

Listing 49 – Example code of SDO confirmation function for Client SDO in Slave application.

```
void csdo_up_con(co_csdo_t* sdo, co_unsigned16_t idx,
                 co_unsigned8_t subidx, co_unsigned32_t ac,
                 const void* ptr, size_t n, void* data) {
  if (ac != 0) {
    printf("CSDO received a non-zero abort code <%#x>\n", ac);
    return;
  }

  co_unsigned32_t val = 0u;
  const uint_least8_t* buf_ptr = (const uint_least8_t*)ptr;
  const co_unsigned32_t bytes_read =
      co_val_read(CO_DEFTYPE_UNSIGNED32, &val, buf_ptr, buf_ptr + n);

  printf("CSDO received %u bytes representing a value of %u\n", bytes_read,
         val);
}
```

If everything goes smoothly a similar to the output from Listing 50 is expected. However, if for example the Master application is started after the SDO request was sent, a “Resource not available: SDO connection” (0x060A0023) abort code is passed to the confirmation function – see Listing 51.

Listing 50 – Example output from Slave application running Client SDO service.

```
SEND <0x601> <0> DATA: 0x40 0x00 0x30 0x00 0x00 0x00 0x00 0x00
RECV <0x581> <0> DATA: 0x43 0x00 0x30 0x00 0x7a 0xf1 0xd8 0x60
CSDO received 4 bytes representing a value of 1624830330
```

Listing 51 – Example error report from Slave application running Client SDO service.

```
CSDO received a non-zero abort code <0x60a0023>
```


ESA Contract No. 4000143237/23/NL/AS

12 Analytical Index

N/A

ESA Contract No. 4000143237/23/NL/AS

13 Lists

13.1 List of Tables

13.2 List of Figures

13.3 List of Listings

ESA Contract No. 4000143237/23/NL/AS