CANopen Library Toolset

Test Suite – Software User Manual

CAN-N7S-CTSDP-SUM rev. 2.2

N7 SPACE SP. Z O.O.

Prepared by	Date and Signature
Konrad Grochowski	[illegible] Konrad Grochowski 2025.09.08 22:11:20 +02'00'
Verified by	
Mateusz Dyrdół	[illegible] Mateusz Dyrdół 2025.09.09 13:02:53 +02'00'
Approved by	
Seweryn Ścibior	Seweryn Ścibior Digitally signed by Seweryn Ścibior Date: 2025.09.09 15:47:58 +02'00'

ESA Contract No. 4000143237/23/NL/AS

Table of Contents

Change Record

Issue	Date	Change
1.0	2021-06-28	Initial release
1.1	2021-10-17	Fixes for CDR RIDs: • word 'optional' replaced with 'alternative' in some sentences • Captions added to all code listings • Explicit mention the Ubuntu 20.04 as the reference system • RD8 reference changed to SCons user manual • 9.2.1. – added note about git submodules
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	2024-11-30	CANopen Library Toolset project MTR: • Document identifier changed from CAN-N7S-UM-21002 to CAN-N7S-CTSDP-SUM • New ESA contract identifier added to footer • Introduction updated • Reference documents updated • Updated handling multiple platforms
2.1	2025-06-04	Release for TRR: • Opening MPLAB moved to CANDP-SUM • Added notes about building LEON3 docker image • Reference documents updated
2.2	2025-09-08	Release for CDR/QR: • Reference documents updated

1 Introduction

This document provides Software User Manual for the CANopen SW Library Test Environment and Test Suite deliverables 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.

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 Test Suite – Interface Control Document	CAN-N7S-CTSDP-ICD	2.4
RD4	CANopen Library Toolset Test Suite – Software Requirements Specification	CAN-N7S-CTSDP-SRS	2.3
RD5	CANopen Library Toolset Test Suite – Software Design Document	CAN-N7S-CTSDP-SDD	2.3
RD6	CANopen Library Toolset Test Suite – Software Configuration File	CAN-N7S-CTSDP-SCF	2.4
RD7	CANopen Library Toolset Coding Standards and Tools	CAN-N7S-CSTD	1.3
RD8	SCons: A software construction tool	https://scons.org/doc/	

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

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 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 24.04 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/
```

C language source code blocks use the below style:

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

Python language source code blocks use the below style:

```python
nmt = co.nmt(network, device)
assert nmt != None  # must be not None
```

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

```
C Preprocessor directive
C Preprocessor include path
Keywords
NULL
None
Number literal
String literal
Comments
Other
```

5 Purpose of the Software

CTSDP – Technical Data package for the CANopen Test Suite – contains two software items:

- CTESW – CANopen SW Test Environment.
- CTSSW – CANopen SW Test Suite

The main purpose of those items is to provide validation environment and tests for the CANSW – CANopen SW Library, an ECSS-compliant [AD2] C language library providing CANopen [RD1] protocol stack software implementation.

5.1 CTESW

Purpose of the CTESW is to provide a framework for defining, building and executing tests needed to validate CANSW. The tests need to execute on two machines connected with CAN buses. One machine is called Host and is the primary driver of the test – the machine user executing tests directly interacts with. The second machine is called HWTB (Hardware Test Bench) and is a space-grade embedded system representative, with dedicated CAN peripherals. Host machine has x86-64 architecture, HWTB is based on target platform (ARM) development board or TSIM simulator (GR712RC).

From the point of view of the user, CTESW provides two main components:

- Plugins for the *SCons* [RD8] build tool, allowing for convenient definition of the test, building and executing it in a well-designed and user-friendly tool.
- C language library for helping defining applications used by the test.

5.2 CTSSW

CTSSW provides set of validation tests for the CANSW. It is built upon CTESW and uses it to build and execute those tests. Provided tests cover all requirements extracted from ECSS standard for the CANopen protocol stack.

6 External view of the software

Both software items are delivered as archive containing source files, build system configuration files and CANSW version under test. For user convenience CTSSW archive embeds CTESW deliverable, but user can replace it with different version if necessary. User can interact with CTSSW and CTESW via command-line interface (CLI) of the *SCons* tool [RD8].

Details on the composition of the software items, versions etc. can be found in data-pack software configuration file – CANDP-SCF [RD6].

6.1 CTESW

The CTESW directory structure can be described as follows:

```
CTESW/
├── doc/
│   └── CTESW Doxygen files
├── docker/
│   └── Dockerfiles for CTESW containers
├── ld/
│   └── Linker scripts for supported microcontrollers
├── resources/
│   ├── ci/
│   │   └── Continuous Integration support scripts
│   ├── bsp/
│   │   └── HWTB Board Support Package headers and source files
│   ├── build-configs/
│   │   └── Build configuration files for SCons
│   ├── LibCANopen/
│   │   ├── lely-core/ - CANSW repository copy
│   │   └── dcf2dev.sh - dcf2dev wrapper script (Python venv)
│   ├── mplab/
│   │   └── MPLAB project generation scripts
│   ├── n7-core/
│   │   └── N7-Core library headers and source files
│   └── TestFramework/
│       ├── Test Framework headers and source files
│       └── HAL/
│           └── Hardware Abstraction Layer implementations
├── site_scons/
│   ├── site_tools/
│   │   ├── libs/
│   │   │   └── Python utilities to be used by SCons plugins
│   │   ├── platforms/
│   │   │   └── SCons settings for supported platforms
│   │   └── SCons plugins sources
│   └── site_init.py - SCons tool extension point
├── sonar/
│   └── Sonar reports generator scripts
├── svf-configs/
│   ├── Software validation test setups configuration files
├── tests/
│   └── CTESW integration tests sources
├── validation/
│   └── CTESW design reviews, inspections and manual tests
├── ctsdp-srs.json - CTESW and CTSSW requirements
└── SConstruct - SCons main definition file for the CTESW
```

6.2 CTSSW

The CTSSW directory structure can be described as follows:

```
CTSSW/
├── configs/ -> environment/configs (symbolic link)
├── environment/
│   └── Copy of the CTESW code
├── resources/ -> environment/resources (symbolic link)
├── site_scons/ -> environment/site_scons (symbolic link)
├── tests/
│   ├── dcf2dev/
│   │   └── dcf2dev validation tests
│   ├── ecss-time/
│   │   └── CANSW ECSS TIME support validation tests
│   ├── emcy/
│   │   └── CANSW EMCY service validation tests
│   ├── nmt/
│   │   └── CANSW NMT service validation tests
│   ├── pdo/
│   │   └── CANSW PDO service validation tests
│   ├── sdo/
│   │   └── CANSW SDO service validation tests
│   └── sync/
│       └── CANSW SYNC service validation tests
├── validation/
│   └── CANSW design reviews, inspections and manual tests
├── candp-srs.json - CANSW requirements
├── candp-sss.json - ECSS standard requirements for CANSW (for SRS tracing)
└── SConstruct - SCons main definition file for the CTSSW
```

7 Operations environment

7.1 General

CANSW validation requires two machines connected with two CAN buses. The first machine, called Host, must be an x86-64 machine running Linux operating system with preinstalled required software (see section 7.3). It also has JTAG and CAN-USB dongles connected to its USB ports and their drivers properly installed. If direct connection to USB ports is troublesome – a proxy machine can be used, which will forward JTAG and CAN messages via Ethernet. Such configuration also allows Host to become embedded as Docker container, which greatly simplifies software configuration process. See next section for better description of supported hardware configuration.

The second machine required for the validation is called HWTB and is based on target platform development board, if necessary (for ARM SAMV71), modified to provide CAN peripheral.

For LEON3 targets it is possible to perform the validation on the simulator (TSIM) alone.

7.2 Hardware configuration

Figure 1 presents general overview of hardware configuration required to execute tests using CTESW (so all tests from CTSSW). This was the configuration used in the activity. It requires additional proxy (as simple as Raspberry PI on the figure) to handle USB drivers for CAN and JTAG dongles, but as a benefit all other software items can be embedded inside Docker container and easily updated during the scope of the project, or reproduced on a different machine.

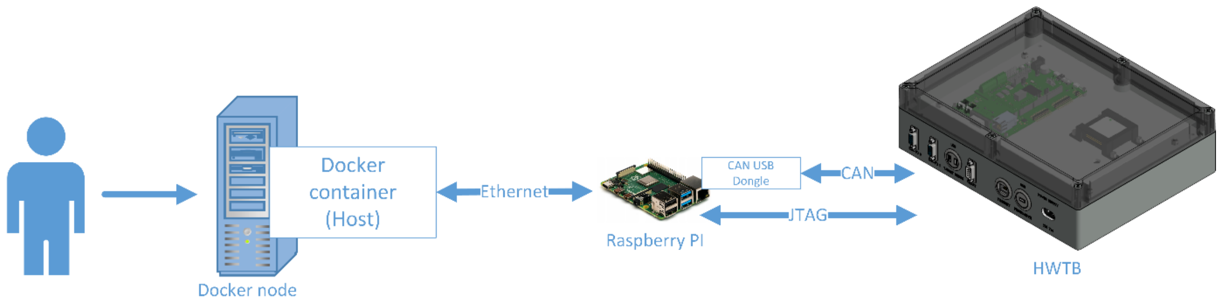

Figure 1 – CTSSW Docker based hardware configuration.

Figure 2 presents alternative configuration, which does not require any proxy – all peripherals are directly connected to physical machine. It requires proper permissions on the Linux machine and proper configuration of all software items on that machine.

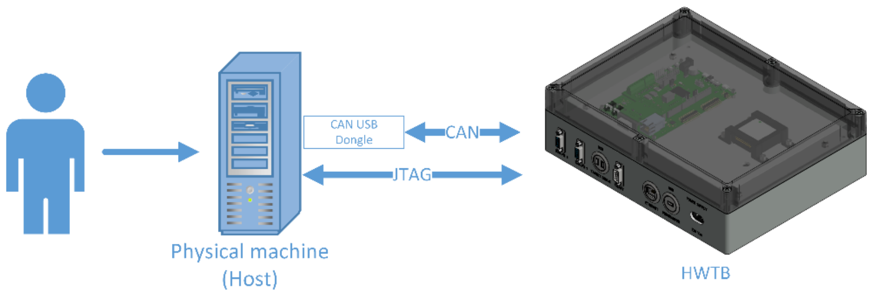

Figure 2 – CTSSW hardware configuration with dedicated physical machine.

For LEON3 platform – GR712RC – an TSIM3 simulator can be used. No external HWTB is required then, only software license for TSIM3. This approach was used in the activity.

7.3 Software configuration

CANSW (SW Under Test) is embedded inside CTESW. CTESW itself is embedded inside CTSSW. This makes CTSSW a standalone application.

Embedded versions must be compatible – so CANSW in version 3.4.x is embedded in CTESW 3.4.x and CTSSW 3.4.x. Delivered packages have all items in proper versions, but user might want to choose a different setup.

CTESW and CTSSW are delivered in the form of source files, so they require proper configuration of the operating system to build and execute tests. Complete tools list can be found in CSTD [RD7]. Simplest approach is to provide only a Docker on Linux and reproduce environment using the container provided with CTESW (as Dockerfile – configuration file – and whole image).

7.4 Operational constraints

CTSSW and CTESW do not provide any operational modes. Only known constraint: due to nature of CAN bus, only single set of tests can be executed at a given time at a given hardware (no parallel connections can be made). So only a single call to CTESW/CTSSW connected to a given HW can be made at once.

8 Operations basics

The main purpose of the CTSSW is to execute (using CTESW) set of validation tests of the CANSW. This is the only operation supported by the CTSSW software. It is divided into smaller steps described in the next chapter, but in general SW supports only one operation and mode – to perform tests.

User commanding is required for CTSSW to start execution. No user interaction is required during tests execution; user needs only to check results when CTSSW finishes operation.

User interface is based on Command Line Interface (CLI) of the SCons tool [RD8].

9 Operations manual

9.1 General

CTESW is a framework for creating and running tests, user mostly interacts with CTSSW itself, hence this chapter will focus on CTSSW operations. User is not expected to execute CTESW itself for any other action then performing its self-tests.

9.2 Set-up and initialization

9.2.1 Obtaining source

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

Listing 1 – Unpacking CTSSW source from ZIP file.

```
$ unzip CAN-N7S-CTSDP-CTSSW-v3_4_0.zip  # assuming version 3.4.0
```

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

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

```
$ tar -xvf unzip CAN-N7S-CTSDP-CTSSW-v3_4_0.tar.bz2  # assuming version 3.4.0
```

Alternatively, CTSSW source can be accessed using publicly available code repository by executing the commands from Listing 3 (assuming version 3.4.0 of the CTSSW).

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

```
$ git clone https://gitlab.com/n7space/canopen/test-suite.git --depth=1 --branch=v3.4.0
$ cd test-suite
$ git submodule update –recursive –init
```

Important note: Git repository of CTSSW uses *git submodules* to reference CTESW repository and CTESW has CANSW in submodule. Hence the additional command in Listing 3. It does not impact the behaviour of delivered archives – they contain full source, including all submodules (even the `*-git-*.tar.bz2` archive, delivered as documentation of software development process).

9.2.2 Setting up the environment

Using Docker is the easiest way to reproduce necessary software environment. Otherwise, user needs to install all dependencies from CSTD [RD7], using operating-system specific packages, which is out of the scope of this document.

Listing 4 uses the Docker image provided as deliverable (it might take minutes to perform the import).

Listing 4 – Importing CTESW Docker image.

```
$ docker image load –input CAN-CTSDP-CTESW-docker-<image>-v3_4_0.tar.bz2
Loaded image: <image>:v3.4.0
```

Where `image` should be replaced with one of the following:

- `core` – Base image used by all other images, useful only when working only with x86 machine.

- `cortex-m7` – Image used for building code for all supported Cortex-M7 platforms (SAMV71, SAMRH71 and SAMRH707). Depends on core image.
- `xc32` – Base image with XC32 toolchain, used by platform-specific XC32 images. Depends on `cortex-m7` image.
- `xc32-ATSAM` *platform* – Images used for building code for XC32-supported platforms. Depends on `xc32` image.
- `leon3-base` – Base image with BCC toolchain, but without TSIM3 simulator (license limits redistribution of the simulator), used by LEON3 images.

User can import only a single image (and this is a recommended action when working with selected platform – the same image can be used for Host and HWTB activities). When importing multiple images it is recommended to start with *core* and then go "bottom up" through dependencies – this should improve disk usage and import speed.

Alternatively, image can be built „from scratch" (assuming all packages are still available) using Dockerfile provided in CTSSW source, as in Listing 5. Depending on the number of dependent images, this steps might be necessary to be repeated (e.g.: build *core, cortex-m7* and *xc32* – in that order).

Listing 5 – Building CTESW Docker custom image.

```
$ cd <path/to/ctssw/source>/environment/docker/core
$ docker build -t n7s/ctesw:v3.4.0-core .  # assuming version 3.4.0
$ cd ../<image_directory>
$ docker build \
    --build-arg REGISTRY=n7s/ctesw \
    --build-arg CORE_IMAGE_TAG=v3.4.0-core \
    -t n7s/ctesw:v3.4.0-<image> .  # assuming version 3.4.0
```

Building your own image is required for LEON3 platform – user must have an existing TSIM3 license and copy to execute tests. Base image with TSIM3 and BCC toolchain should be provided via `REGISTRY` and `IMAGE_TAG` arguments when building `leon3` container from it's `Dockerfile`.

User might also download image directly from publicly available Docker container registry, by providing `registry.gitlab.com/n7space/canopen/test-environment:v3.4.0-<image>` as the image name to `docker run` command.

After setting up the image, user might use Docker containers as in Listing 6 (substituting *image* with name of the image appropriate for the chosen HTWB platform).

Listing 6 – Executing command in CTESW Docker container.

```
$ docker run --rm -v $PWD:$PWD -w $PWD -u $(id -u):$(id -g) n7s/ctesw:v3.4.0-<image> <COMMAND>
```

This command will mount current directory and execute container with privileges of the current user. It is recommended to call it this way always in the root of the CTSSW source directory.

It can be very convenient to set up this command as an alias in Linux shell as in Listing 7. This will allow for a quick execution of other commands inside containers.

Listing 7 – Shell alias for executing command in CTESW Docker container.

```
$ alias docker-here='docker run --rm -v $PWD:$PWD -w $PWD -u $(id -u):$(id -g)'
```

For example, to check correctness of the image and CTSSW source, user might execute commands like in Listing 8 (or without alias as in Listing 9) and expect similar output.

All following commands in this chapter assume that there are either executed on properly configured environment, or are proceeded with `docker run` / alias.

Listing 8 – Example command executed in CTESW Docker container.

```
$ cd <path/to/ctssw/source>
$ docker-here n7s/ctesw:v3.4.0-core scons -h
scons: Reading SConscript files ...
# ...
# ... other help lines ...
# ...
CTSSW - CANopen SW Library Test Suite - v3.4.0
Licensed under European Space Agency Public License (ESA-PL) Permissive (Type 3) – v2.4
Copyright N7 Space sp. z o.o. 2020-2025
# ...
```

Listing 9 – Example command executed in CTESW Docker container.

```
$ cd <path/to/ctssw/source>
$ docker run --rm -v $PWD:$PWD -w $PWD -u $(id -u):$(id -g) n7s/ctesw:v3.4.0-core scons -h
# same output as in Listing 8
```

9.2.3 Configuration

CTSSW needs to be configured to correctly work in a given hardware configuration. There are two configuration files required by `scons` in order to build and run the tests – build configuration and SVF (runtime) configuration.

9.2.3.1 Runtime configuration

Runtime configuration for HWTB is stored in INI file, some examples are available in `svf-configs/` subdirectory of CTESW. By default `localhost.conf` file is loaded, and user must edit it or select an appropriate file for running the tests on development setup. Listing 10 provides contents of an example configuration file.

Listing 10 – CTESW configuration file example (`example.conf`).

```
# GDB client configuration (running on host)
[gdb]
path = arm-none-eabi-gdb
verbose = True
customResetCmd = monitor reset
    monitor reset 0
    monitor reset 1
    monitor reset 8
    monitor reset
delayStartupCommands = True
enableLowPowerHandlingMode = True
timeout = 120

# GDB server configuration (running on RaspberryPi or host, depending on hardware configuration)
[gdbServer]
address = canopen-rpi
```

```
port = 2331
username = arm
password = armdev
path = /opt/SEGGER/JLink_V780/JLinkGDBServerCLExe
args = -select USB -device ATSAMV71Q21 -endian little -if swd -speed 4000 -noir
verbose = True
connectedFlag = J-Link is connected
failedFlag = Could not connect to J-Link

# HWTB power control configuration
[egse]
address = canopen-rpi
username = arm
password = armdev
power_control = none

# HWTB I/O handler selection, currently only RTT is supported
[ioHandler]
type = rtt

# HWTB RTT I/O handler configuration
[ioHandler.rtt]
address = canopen-rpi
port = 5555
username = arm
password = armdev

# CAN bus A configuration
[canBusA]
address = canopen-rpi
username = arm
password = armdev
ipLinkCanId = can0
port = 6500
mcanId = 0
verbose = True
ipLinkCanConfig = bitrate 1000000 fd off sample-point 0.875

# CAN bus B configuration
[canBusB]
address = canopen-rpi
username = arm
password = armdev
ipLinkCanId = can1
port = 6600
mcanId = 1
verbose = True
ipLinkCanConfig = bitrate 1000000 fd off sample-point 0.875
```

Highlighted lines are usually the only ones requiring user attention. They contain address and credentials needed to access the proxy connected to HWTB. In configuration without proxy, `localhost` needs to be provided as address (`username` and `password` are not necessary then). Other options require changing only when operating with different hardware configuration than HWTB.

9.2.3.2 Build configuration

Build configuration files for HWTB are stored in `resources/build-configs` directory of CTESW. This file should contain definitions for building the code for specified target board. The contents of this file depends on the platform, as most of the definitions from it are used in HAL and BSP code. By default, `default-<platform>.conf` is loaded. The simplest way of creating a new tailoring for a custom board is copying the default file for a platform, and modifying the values of the variables to match the hardware configuration of the microcontroller on target board. Changes in those files are necessary only when using custom boards (other than supported development boards).

9.2.4 Checking the configuration

After setting up and configuring CTSSW and HWTB it is recommended to validate the configuration by running CTESW self-tests. This requires optional *cram* tool to be present (available in the CTESW Docker image). Those tests perform various checks of the CTESW setup. They are expected to take 20-90 min (depending on the Host hardware).

To execute them, go to CTESW subdirectory in CTSSW and execute commands (for example inside environment provided by the Docker image) from Listing 11 and expect similar output. `platform` should be replaced with the name of a supported platform (for example, `samv71q21`), and `devboard` should be replaced with the name of SVF configuration file for selected development board. Configuration is read from `svf-configs/<platform>/<devboard>.conf` file.

Listing 11 – Commands to execute CTESW validation tests.

```
$ cd <path/to/ctssw/source>/environment
$ ./run-cram.sh <platform> <devboard>
# …
ALL TESTS PASSED!
```

Any other message than `ALL TESTS PASSED!` means that CTESW or HWTB is not setup or configured properly and requires investigation. Logs from tests execution can be found in `environment/build/release/tests` subdirectory. Reading messages provided in those logs should help diagnose the issue. Most probable problems are the ones related to connection configuration and access to proxy.

After making changes to the CTSSW configuration, re-execution of tests is recommended. It is highly recommended to remove `environment/build/release/tests` subdirectory before running the tests again (or at least subfolder containing output of the failing test). Otherwise, the failed tests won't be rebuilt if ran via `cram`, and will keep being reported as failed instead.

To help investigation, `run-cram.sh` script accepts the list of test paths to run as an argument, so not all tests needs to be re-run each time.

9.3 Getting started

After setting up and validating the CTESW as described in previous section, there are no other actions to be performed – user can execute the CANSW test suite.

9.4 Mode selection and control

N/A

9.5 Normal operations

Execution of the whole test suite is simply done by calling the SCons tool - Listing 12.

Listing 12 – Commands to execute CTSSW validation tests.

```
$ cd <path/to/ctssw/source>/
$ scons hwtbPlatformName=<platform> svf=<path/to/svf/file>
```

Or by using Docker – Listing 13.

Listing 13 – Commands to execute CTSSW validation tests inside Docker container.

```
$ cd <path/to/ctssw/source>/
$ docker run --rm -v $PWD:$PWD -w $PWD -u $(id -u):$(id -g) n7s/ctesw:v3.4.0-<image> \
        scons hwtbPlatformName=<platform> svf=<path/to/svf/file>
```

For clarity, further commands examples will no longer include `docker run ...` prefix, it is up to the user to choose how the `scons` command is called.

The selected Docker image (*image* in the examples) must be appropriate for the selected platform (*platform* in the examples). The chosen SVF must provide the given platform – CTSSW does not have the means to detect the hardware type, so it is responsibility of the user to select those arguments properly.

To speed up the tests execution process, on multi-core platforms it is recommended to compile applications used by the tests before executing the tests themselves. This can be achieved by passing special target to the SCons (`test-cases-apps`) and selecting desired parallel jobs count. For example, Listing 14 uses 10 parallel jobs.

Listing 14 – Building `scons` target using multiple jobs.

```
$ scons hwtbPlatformName=<platform> svf=<path/to/svf/file> \
       -j 10 test-cases-apps unit-tests-apps
```

Tests themselves **cannot** be executed in a parallel fashion (`-j` must be equal to 1 when running them).

SCons by default terminates the execution at the first occurrence of error. If user wants to try to perform all tests, even if some of them fail, the `-k` switch should be added to the `scons` call.

Test execution can take 40 – 120 minutes, depending on the Host hardware.

During the execution SCons prints logs of the performed operation (including build commands, GDB commands, CAN bus data exchange, etc.). If log needs to be archived it is recommended to use `tee` command, to keep seeing progress on the standard output – Listing 15.

Listing 15 – Using `tee` to observe and store build logs at the same time.

```
$ scons <svf options> <targets> 2>&1 | tee ctssw.log
```

9.6 Normal termination

After all tests pass, the call to `scons` should end with `0` (zero) return code and the message:

```
scons: done building targets.
```

If that message is not present near the end of the output (it might be followed with some clean-up messages, depending on the network speed) something went wrong and not all tests have passed.

Logs from execution of the tests can be found in `build/release/tests` subfolder. Each test produces the following logs:

- Output from application executed on Host (`<test name>.host.log`),
- Output from application executed on HWTB (`<test name>.hwtb.log`),
- CAN messages exchanged on bus A (`<test name>.can-a.log`),
- CAN messages exchanged on bus B (`<test name>.can-b.log`),
- Dummy file present only when the test passes (`<test name>.log`).

Test name is in the form: `<test case name>-<direction>` where direction is either `host-to-hwtb` or `hwtb-to-host`. This is caused by each test case being "symmetrical" so each test application is executed once on Host and once on HWTB. This means that each test case specified in CTSSW is executed twice during the test suite run.

9.7 Error conditions

In case of any test failure the call to `scons` should end with non-zero return code and the message:

```
scons: building terminated because of errors.
```

It should be preceded with one or more messages like:

```
scons: *** [build/release/tests/<test name>/<log file name>] Error <error code>
```

It is a suggestion where to look for the error information. Error code is platform dependent and should not be investigated. Logs should be available for investigation – see previous section for details.

In case of an error on earlier stage (build, linking etc.) error message should be present directly in the SCons output.

9.8 Recover runs

Before re-running the tests it is recommended to remove `build/release/tests` folder. Or at least its subfolder containing output from the failing test. In the latter case SCons will try to execute only the tests that were not successful. User might also execute selected subset of tests by passing their names as targets to `scons` call (removing output from failing test still might be necessary).

10 Reference manual

10.1 Introduction

All information necessary to execute CTSSW test suite and to validate CANSW can be found in section 9. This chapter provides some helpful information if user would like to customize the execution process or debug problems.

Detailed reference of all available functions of the CTESW can be found in the CTSDP ICD [RD3]. Details on using SCons can be found in its documentation – [RD8].

10.2 Help method

CTSSW and CTESW provide simple help method, available when calling `scons -h` in root folder of the selected software. As shown below, it lists available options and their current options:

```
$ scons -h
scons: Reading SConscript files ...
Detected SW version from git: v3.4.0
Mkdir("build/release/")
Mkdir("build/release/gcc_host")
Mkdir("build/release/samv71q21")
scons: done reading SConscript files.

build: Defines build type (release|debug|coverage)
    default: release
    actual: release

svf: SVF hardware configuration file ( /path/to/svf )
    default: svf-configs/localhost.conf
    actual: svf-configs/localhost.conf

checkCode: Set to enable additional code checks (yes|no)
    default: True
    actual: True

optimization: Optimization option used by release build (g can be passed to debug and
coverage builds also) (g|0|1|2|3|s)
    default: 2
    actual: 2

disableAsserts: Disables assertions in the code (smaller binary but reduced memory
corruption detection) (yes|no)
    default: False
    actual: False

buildDirName: Name of the main build directory (usefull for parallel CI builds)
    default: build
    actual: build

chipConfig: MCU configuration file ( /path/to/chipConfig )
    default: resources/build-configs/default.conf
    actual: resources/build-configs/default.conf

hwtbPlatformName: Defines HWTB platform name (samv71q21|samrh71f20|samrh707f18)
    default: samv71q21
    actual: samv71q21


CTESW - CANopen SW Library Test Environment - v3.4.0
Licensed under European Space Agency Public License (ESA-PL) Permissive (Type 3) – v2.4
Copyright N7 Space sp. z o.o. 2020-2024


Use scons -H for help about SCons built-in command-line options.
```

Last line of the above message informs user about a way of getting SCons general options:

```
$ scons -H
```

10.3 Screen definitions and operations

N/A

10.4 Commands and operations

Basic commands are provided in section 9.5. As mentioned there, CTSSW provides only a single command – `scons` – and it is all that is needed to perform the suite. User might want to execute a single test, this can be achieved by passing its name to SCons like `scons test-name`. List of all test names can be obtained from CANSW documentation, or by calling `scons traces` and browsing JSON file `build/release/cansw-traces.json` containing list of all available tests along with their documentation. SCons also accepts some switches that change the way the test suite is executed, see Table 1 for details.

Table 1 – CTSSW SCons options.

Option	**Description**
CTSSW specific	
`svf`	Path to configuration file, as described in 9.2.3.1. Detailed options available in Table 2.
`build`	One of `release/debug/coverage`. In normal test run only release should be used. Debug mode can be used to build application if detailed debugging is needed (but test might fail in this mode due to performance degradation). Coverage mode could be used to obtain line and branch coverage information, but in most cases it introduces too big execution and size overhead – this mode is recommended only for CTESW developers.
`checkCode`	Setting this to "no" will disable code checking during build process. Useful for speeding up the builds during development, checks may take considerable amount of time.
`optimization`	Optimization option. For `release` builds, it can be `g`, `0`, `1`, `2`, `3` or `s`. For `debug/coverage` builds, it can be either `g` or `0`.
`disableAsserts`	Disables the asserts in the code. This may reduce the binary size, but will reduce the amount of runtime checks performed.
`buildDirName`	Name of the build directory (useful when performing various builds from the same source directory)
`chipConfig`	Chip configuration file with definitions of tested platform's MCU configuration variables, as described in 9.2.3.2.

Generic SCons options	
`-k`	"Keep going" – continue execution after error occurrence. Useful to gather information about all failing tests.
`-j <N>`	Execute in `N` parallel jobs. See notes in 9.5. Do not use for executing tests.
`--debug=explain`	SCons will display the reason for rebuilding a given target. Useful while debugging failing tests or while developing a new tests.

Table 2 – CTSSW configuration file options.

Option	**Description**
`[gdb]`	
`path`	Path to GDB executable.
`verbose`	`True/False` – When set, GDB will include additional information in SCons log (recommended).
`customResetCmd`	Custom reset commands for the MCU. Multi-line strings are supported.
`delayStartupCommands`	If `True`, all the commands executed before platform startup will be queued for execution after the startup is performed.
`enableLowPowerHandlingMode`	For JLink SEGGER, setting this to `True` results in less strict handling of communication errors, which may led to stability increase when resetting the MCU.
`timeout`	Timeout for GDB commands execution
`[gdbServer]`	
`address`	Address of the proxy to run GDB Server on, or `localhost` to run locally.
`port`	TCP port to run GDB server on.
`username`	Credentials required to access proxy via SSH.
`password`	
`path`	Path on the proxy (or local) to GDB Server executable.
`args`	GDB Server executable command line arguments.
`verbose`	`True/False` – When set, GDB server will include additional information in SCons log (not recommended, usually all interesting information is reported by GDB, setting to true might slow down proxy operations).
`timeout`	Timeout for GDB communication
`connectedFlag`	String written to standard output by the debugger that indicates the microcontroller has successfully connected to the debugger
`failedFlag`	String written to standard output by the debugger that indicates it couldn't connect to the microcontroller.
`[egse]`	
`address`	Address of the proxy controlling the HWTB power supply, or `localhost` to run locally.
`username`	Credentials required to access proxy via SSH.
`password`	
`power_control`	Power control type used by the proxy board. Can be `pin`, `command` or `none`. When option other than `none` is selected, an `[egse.power.<option>]` section must appear in this file, with appropriate configuration options.
`[egse.power.pin]`	
`pin`	Pin used to control board power.

Option	Description
`onIsHigh`	If `True`, the setting `pin` to high state powers on the board.
`[egse.power.command]`	
`onCmd`	Command to execute in order to enable HWTB power supply.
`offCmd`	Command to execute in order to disable HWTB power supply.
`[ioHandler]`	
`type`	Selection of the way tests applications should handle their standard output: • `rtt` – Standard output is transmitted in real-time through Segger's RTT protocol using debugger's link, During CANSW validation, the RTT option is used by default, to limit necessary links to the HWTB.
`[ioHandler.rtt]`	
`address`	Address of the proxy to run JLink RTT server, or `localhost` to run locally.
`username`	Credentials required to access proxy via SSH.
`password`	
`port`	TCP port used by JLink RTT server.
`[canBusA]/[canBusB]`	
`address`	Address of the proxy to run `socat` CAN forwarder, or `localhost` to run locally.
`username`	Credentials required to access proxy via SSH.
`password`	
`ipLinkCanId`	Linux CAN interface id (`ip link` command) to be used by given bus. Identifier must be present on the proxy (or locally).
`port`	Port to be used by `socat` to make CAN available over Ethernet. Used even when setup locally.
`mcanId`	MCAN device identifier to be used by given bus on the HWTB.
`verbose`	`True/False` – When set, CAN traffic is visible in SCons log. Recommended for default bus (A), not recommended to set for both buses at once (log becomes unreadable).
`ipLinkCanConfig`	Arguments for the `ip link` command used to configure the CAN link for communication with HWTB

10.5 Error messages

As described in 9.7, SCons provides a single type of message when the command execution failed. To investigate the reason of the failure, user must look through previous log messages from SCons, or into detailed logs provided by the test itself. Messages in SCons log can include messages from operating system (regarding network connection problems), used application (GCC compilation problems, GDB errors etc.) and tests themselves.

Messages provided in output logs from applications are test-specific (test's author is free to provide any message), but all are prefixed with timestamp since the test execution start and all logs should end with message containing exit code of the application – if it's missing, the application has crashed.

11 Tutorial

11.1 Introduction

Chapters 9 and 10 provide complete introduction to CTSSW usage, including a step-by-step tutorial. This chapter is focusing on CTESW and provides information for the user who would want to create new dedicated tests for the CANSW.

11.2 Getting started

CTESW is a framework for creating CANSW tests. It consists of two main components: SCons extensions used to specify the test for the build tool and Test Framework – a C language library for creating tests application that will use and validate CANSW features. This tutorial will show, how to use those components to create user own test.

11.3 Using the software on a typical task

11.3.1 Add new test to CTSSW

11.3.1.1 Select folder for the test

This is the optional step, when the tests do not match any existing categories of tests. It is however a recommended step for “project specific” tests.

All tests are stored inside `tests/` CTSSW subdirectory. New folder needs to be added there. Then, it needs to be added to the main SCons configuration file in the root CTSSW directory – `SConstruct`. It already contains a list of used directories, new one needs to be added to it like in Listing 16. Then the newly created folder needs its own SCons configuration file – `SConscript`, see Listing 17 for example. It contents will be filled in the next stages of this tutorial.

Listing 16 – Example modification of `SConstruct` to add new tests folder.

```
tests = [
    "tests/dcf2dev",
    "tests/emcy",
    "tests/ecss-time",
    "tests/nmt",
    "tests/pdo",
    "tests/sdo",
    "tests/sync",
    "tests/NEW-TEST-FOLDER-NAME",
]
```

Listing 17 – Empty `SConscript` add for new tests.

```
Import("env")

tests = []

# here the folder specific test will be added

env.Alias("NEW-TEST-GENERIC-NAME-tests", tests)  # not required, suggested for convenience
Return("tests")
```

11.3.1.2 Define two device descriptions used by the test

This is an optional step – user might want to define device's Object Dictionaries manually using CANSW API. It is more convenient however to define them using DCF (Device Configuration File) format from CiA 306 standard. DCF definition is out of the scope of this document. After preparing two matching Object Dictionary definitions, with required services etc. configured, user should place two `.dcf` files inside test folder.

11.3.1.3 Write C code of both test applications

Main test code will go inside C application, one running on Host, second on HWTB. Both should follow the same scheme, shown in Listing 18.

Listing 18 – Example of C source file of new tests.

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

#include <lely/co/SERVICE-TO-BE-TESTED.h>

#include <TestFramework/TestHarness.h>

void
TestSetup(can_net_t *const net)
{
	// procedure called once, before the test start
	// should be used to initialize the service

	// Example:

	dev = dcf_sdo_abort_transfer_client_init();  // code from DCF file

	csdo = co_csdo_create(net, dev, SDO_NUM);

	if (co_csdo_start(csdo) != 0)
		FAIL_TEST("SSDO service start failed");
}

void
TestTeardown(void)
{
	// procedure called once, after the test finishes
	// should be used to clean up services

	// Example:
	co_csdo_stop(csdo);
	co_csdo_destroy(csdo);
}

void
TestMessageReceived(const struct can_msg *const msg)
{
	// Procedure called for each received message on CAN bus A.
	// It is called after lely-core processed the message.
}
```

```
void
TestStep(void)
{
	// Procedure called in the loop while the test is performed.

	// Example:
	step++;
	if (step > STEP_COUNT)
		FINISH_TEST();
}

void
TestMessageSent(const struct can_msg *const msg)
{
	// Procedure called for each message sent by lely-core on CAN bus A.
}
```

11.3.1.4 Add test specification to SCons configuration

When all tests files are ready, user must add them to proper `SConscript` – either existing one or the newly created one from 11.3.1.1.

Inside it a template like it should be filled with proper names of the files – see Listing 19.

Listing 19 – Part of example `SConscript` with new test added.

```
dcfApp1 = env.Dcf2Dev("app1.dcf")
dcfApp2 = env.Dcf2Dev("app2.dcf")

tests += env.MakeSymmetricalTestCase(
    "NAME-OF-THE-TEST",
    ["app1.c"] + dcfApp1,
    ["app2.c"] + dcfApp2,
    trace={  # optional block of documentation, whole parameter can be omitted
        "title": "HUMAN READABLE TEST TITLE",
        "traces": ["REQUIREMENT-1", "REQUIREMENT-2"],
        "doc": {
            "given": "INPUTS",
            "when": "TESTED FEATURE",
            "then": "OUTPUTS",
        },
    },
)
```

11.3.1.5 Run the test

When the test specification is ready it is time to run it and see if everything is correct:

```
$ scons hwtbPlatformName=<platform> svf=<path/to/svf/file> NAME-OF-THE-TEST
```

Execution should finish normally. Possible errors will be reported in SCons log.

11.3.2 Generating MPALB X IDE project for CANSW

CTESW provides utilities for generating MPLAB X IDE library projects with lely-core sources. Those projects are generated from lely-core's `compile_commands.json` file, which is automatically created by `scons`. The configuration of lely-core (`./configure` flags, etc.) in generated project is exactly the same as in CTESW/CTSSW. This means that the generated project uses the same build flags as CTESW, so in order to generate MPLAB project for `debug` builds, it should be explicitly specified via `scons` arguments.

In order to create compile_commands.json file for lely-core, run the following command:

```
$ scons hwtbPlatformName=<platform> svf=<path/to/svf/file> LibCANopen
```

Afterwards, to generate MPLAB project run the Python script included in `resources/mplab` directory of CTESW. This script does not have any external dependencies. To list the options of the script, use `--help` argument.

```
$ python resources/mplab/generate_mplab_project.py --help
usage: generate_mplab_project.py [-h] [--lely-core-dir LELY_CORE_DIR] [--makefile-path
MAKEFILE_PATH] [--lely-config-path LELY_CONFIG_PATH] compile_commands_path project_path
{atsamv71q21b,atsamv71q21rt,atsamrh71f20c}

Generate MPLAB project

positional arguments:
  compile_commands_path
                        Path to compile_commands.json file
  project_path          Directory of generated project
  {atsamv71q21b,atsamv71q21rt,atsamrh71f20c}
                        Target platform

options:
  -h, --help            show this help message and exit
  --lely-core-dir LELY_CORE_DIR
                        Path to lely-core sources, auto-detected if not provided
  --makefile-path MAKEFILE_PATH
                        Path to custom Makefile. If not provided, Makefile should be placed
next to the script
  --lely-config-path LELY_CONFIG_PATH
                        Path to config.h file. If not provided, it's assumed that it's in
the same directory as compile_commands.json
```

Example invocation of this script that will generate a project in `./libcanopen-mplab` directory in release mode for SAMV71Q21:

```
$ python resources/mplab/generate_mplab_project.py
./build/release/samv71q21/resources/LibCANopen/lely-core/compile_commands.json
./libcanopen-mplab atsamv71q21b
```

12 Analytical Index

N/A

13 Lists

13.1 List of Tables

13.2 List of Figures

13.3 List of Listings