ECSS-E-ST-50-15C Protocol On-Board SW Implementation

Test Suite – Software User Manual

CAN-N7S-UM-21002 rev. 1.3

N7 SPACE SP. Z O.O.

Prepared by	Date and Signature
Konrad Grochowski	
Verified and approved by	
Michał Mosdorf	

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)

1 Introduction

1.1 Purpose, scope and content

This document provides Software User Manual for the CANopen SW Test Environment (CTESW) and the CANopen SW Test Suite (CTSSW). Those software items provide means to perform validation tests of the CANopen SW Library (CANSW) from CANDP.

The following introduction provides a short description of the project objectives.

The CTSDP 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.

1.2 Project motivation and objectives

The use of CAN bus as the main intra-spacecraft communication interface is likely to increase in the coming years. The MIL-STD-1553 standard, which still is one of the most popular interfaces for communication links, has major drawbacks that make it expensive to use or unsuitable for smaller crafts. One such drawback is the complexity of the physical layer. In contrast, the CAN bus is characterized by markedly lower power consumption, design simplicity and reduced complexity, which in turn impacts the size of controller components. With this shift a need arises for a reliable software framework that will allow application software to efficiently exchange data over the bus.

Another reason for the increase in CAN bus-based solutions usage in space applications is the availability of various tools and devices due to ubiquity of CAN related protocols in other industries, mainly automotive and automation. The fairly young ECSS-E-ST-50-15C extension standard does not yet have a mature ecosystem compared both to the CAN-specific domains as well as other space standards. Creating an open source library will help the space industry as a whole to develop and maintain a reusable CAN-based toolset dedicated for space applications. Starting from an existing library is a way to exploit the experience of other industries and to create a dependable library at a possibly lower cost. Because of that, comparison and verification of applicability of available open source CANopen libraries is an important entry task in this study. As a result of that task, a library should be selected as a base of further development.

The ECSS-E-ST-50-15C standard extends the basic CANopen protocol with new features, so the functionality of the selected library will have to be extended. The preferred approach to creating a lasting solution involves providing contributions to an existing open-source project and retaining a single code base. Keeping a single code base would be an important asset, making maintenance more effective and further improvement of the selected library easier in the future. As such, this activity shall involve reaching out to the communities involved in the development of the analysed libraries.

Software dedicated for space industry needs to adhere to strict reliability and safety standards. One of the technical objectives of this activity is to apply proper verification and validation procedures to the selected library required by Category B in accordance with ECCS standards. Library verification process will include using automated tools to statically check correctness of the source code, the coding standard applied, coverage of tests, etc.; those elements should preferably become a part of the normal development process. From potential library's user's point of view, a set of tests to be performed on the target platform is required to check if the library is operating properly on the chosen devices. Those tests should form a validation suite and should be created as a part of this activity, together with a complete test environment needed for their execution.

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	CANopen Application Layer and Communication Profile, CAN in Automation	CiA 301	Version 4.2.0
RD2	CANopen electronic data sheet specification	CiA 306	Version 1.3.0
RD3	ECSS-E-ST-50-15C Protocol On-Board SW Implementation Test Suite - Interface Control Document	CAN-N7S-IF-20002	1.5
RD4	ECSS-E-ST-50-15C Protocol On-Board SW Implementation Test Suite – Software Requirements Specification	CAN-N7S-RS-20003	1.2
RD5	ECSS-E-ST-50-15C Protocol On-Board SW Implementation Test Suite – Software Design Document	CAN-N7S-DD-20002	1.5
RD6	ECSS-E-ST-50-15C Protocol On-Board SW Implementation Test Suite – Software Configuration File	CAN-N7S-TN-20002	1.6
RD7	ECSS-E-ST-50-15C Protocol On-Board SW Implementation HWTB User Manual	CAN-MA-911-001-BDS	1.0
RD8	SCons: A software construction tool	https://scons.org/doc/4.1.0.post1/HTML/scons-user.html	

3 Terms, definitions and abbreviated terms

This document acronyms and abbreviations are listed here under.

BDS	BD Sensors
CAN	Controller Area Network
CANDP	Technical Data package for the CANopen SW Library
CANSW	CANopen SW Library
CLI	Command-Line Interface
CTESW	CANopen SW Test Environment
CTSDP	Technical Data package for the CANopen Test Suite
CTSSW	CANopen SW Test Suite
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 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/
```

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 SAM V71 ARM development board.

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/
├── configs/
│   └── CTESW configuration files (HTWB addresses etc.)
├── docker/
│   └── Dockerfile - CTESW environment container definition
├── resources/
│   ├── LibCANopen/
│   │   └── lely-core/ - CANSW repository copy
│   ├── SAMV7-BSP/
│   │   └── HWTB Board Support Package headers and source files
│   └── TestFramework/
│       ├── HAL/
│       │   ├── Arm/
│       │   │   └── Hardware Abstraction Layer implementation for HWTB
│       │   └── x86/
│       │       └── Hardware Abstraction Layer implementation for Host
│       └── Test Framework headers and source files
├── site_scons/
│   ├── site_tools/
│   │   ├── libs/
│   │   │   └── Python utilities to be used by SCons plugins
│   │   └── SCons plugins sources
│   └── site_init.py - SCons tool extension point
├── tests/
│   ├── design-reviews.json - results of design review validation process
│   ├── inspection-results.json - results of inspection validation process
│   └── CTESW integration tests sources
├── 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/
│   ├── ci-traces/
│   │   └── Notes for CANSW requirements validated via CTSSW CI
│   ├── 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
│   ├── design-reviews.json - results of CANSW design review (validation)
│   └── inspection-results.json - results of CANSW inspections (validation)
├── 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 it is a dedicated, SAM V71 ARM based device, with CAN peripheral. It was developed for this project specific needs, but it is based on standard development board and should be fairly simply replaceable.

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.

HWTB manual can be found in [RD7].

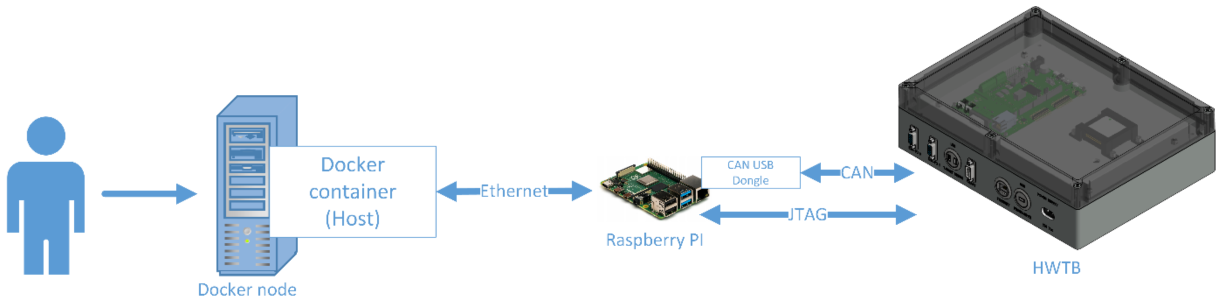

Figure 1 – CTSSW Docker based hardware configuration.

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.

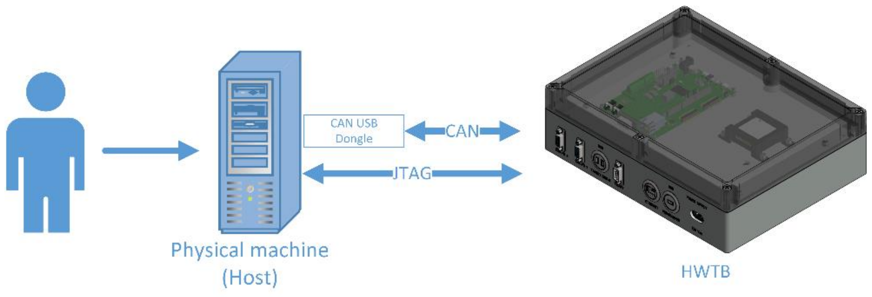

Figure 2 – CTSSW hardware configuration with dedicated physical machine.

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 3.1.x is embedded in CTESW 3.1.x and CTSSW 3.1.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. Table 1 lists required software for CTSSW/CTESW build and execution. 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).

Table 1 – CTSSW execution SW environment.

Tool	Version	Purpose
Container environment		
Docker	19.03.12	Container manager. CTSDP includes image containing all other tools from this table.
Ubuntu	20.04	Operating system (or compatible)
Validation testing environment		
Ubuntu	20.04	Operating system (or compatible)
gcc x86/x64	4.9.0 5.5.0 6.5.0 7.5.0 8.4.0 9.3.0 10.3.0	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.
gcc ARM	arm-none-eabi-9-2020-q2-update-x86_64-linux	GNU C Compiler for ARM platform.
Python	3.8.5	Python language interpreter (test scripts executor)
Autotools	autoreconf 2.69	Build system
SCons	4.1.0	Build system
CppUTest	4.0	Unit test library
paramiko	2.7.2	Python modules (PIP) used by the CTESW to execute tests on the HWTB
pygdbmi	0.9.0.3	
pyserial	3.5	
scp	0.13.3	
timeout_decorator	0.5.0	
cram	0.7	(Optional) Tool for testing CTESW CLI interface

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-CTSDP-suite-src-v3_1_2.zip  # assuming version 3.1.2
```

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-CTSDP-suite-src-v3_1_2.tar.bz2  # assuming version 3.1.2
```

Alternatively CTSSW source can be accessed using publicly available code repository by executing the commands from Listing 3 (assuming version 3.1.2 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=v3.1.2
$ 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 also impacts the behaviour of delivered archives - `*-src-*.tar.bz2` archives contain full source (and is default deliverable of the project), including all submodules, but `*-git-*.tar.bz2` (delivered as documentation of software development process) contains only a snapshot of the Git repository, so submodules must be obtained separately (either by performing proper *git* command or by unpacking required archives).

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 Table 1, 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-docker-v3_1_2.tar.bz2
Loaded image: ctesw:v3.1.2
```

Alternatively, image can be built „from scratch" (assuming all packages are still available) using Dockerfile provided in CTSSW source, as in Listing 5.

Listing 5 – Building CTESW Docker image.

```
$ cd <path/to/ctssw/source>/environment/docker
$ docker build -t ctesw:v3.1.2 .  # assuming version 3.1.2
```

User might also download image directly from publicly available Docker container registry, by providing `registry.gitlab.com/n7space/canopen/test-environment:v3.1.2` as the image name to `docker run` command.

After setting up the image, user might use Docker containers as in Listing 6.

Listing 6 – Executing command in CTESW Docker container.

```
$ docker run --rm -v $PWD:$PWD -w $PWD -u $(id -u):$(id -g) ctesw:v3.1.2 <COMMAND>
```

This command will mount current directory and execute container with privileges of 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 scons -h
scons: Reading SConscript files ...
# ...
# ... other help lines ...
# ...
CTSSW - CANopen SW Library Test Suite - v3.1.2
Licensed under European Space Agency Public License (ESA-PL) Permissive – v2.3
Copyright N7 Space sp. z o.o. 2020-2021
# ...
```

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) ctesw:v3.1.2 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. Configuration is stored in INI file, some examples are available in `configs/` subdirectory. By default

`default_sdram.conf` file is loaded, but user might prefer to choose a different one when calling CTSSW. Listing 10 provides contents of an example configuration file.

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

```
[gdb]
address = canopen-rpi:2331
path = gdb-multiarch
verbose = True

[gdbServer]
address = canopen-rpi
username = pi
password = emeraldpi
path = /opt/JLink_Linux_V680d_arm/JLinkGDBServer
args = -select USB -device ATSAMV71Q21 -endian little -if swd -speed 4000 -noir -vd -timeout 2000
verbose = False

[ioHandlerType]
ioHandlerType = sdram

[sdramIoHandler]
address = canopen-rpi
username = pi
password = emeraldpi
path = /opt/JLink_Linux_V680d_arm/JLinkExe
device = ATSAMV71Q21
speed = 4000
interface = SWD
dataAddress = 0x70000000

[canBusA]
address = canopen-rpi
username = pi
password = emeraldpi
ipLinkCanId = can0
socatPort = 6500
mcanId = 0
verbose = True

[canBusB]
address = canopen-rpi
username = pi
password = emeraldpi
ipLinkCanId = can1
socatPort = 6600
mcanId = 1
verbose = False
```

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.

9.2.4 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 use the `default_sdram.conf` configuration file and 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.

Listing 11 – Commands to execute CTESW validation tests.

```
$ cd <path/to/ctssw/source>/environment
$ ./run-cram.sh
# …
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).

To help investigation `run-cram.sh` script accepts as an argument the list of test paths to run, 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
```

Or by using Docker - Listing 13.

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

```
$ cd <path/to/ctssw/source>/
$ docker run --rm -v $PWD:$PWD -w $PWD -u $(id -u):$(id -g) ctesw:v3.1.2 scons
```

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 default execution assumes that `configs/default_sdram.conf` configuration file is used, but user can pass a different one - Listing 14.

Listing 14 – Passing CTESW configuration file to test execution.

```
$ scons config=<path/to/conf/file>
```

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 15 uses 10 parallel jobs.

Listing 15 – Building `scons` target using multiple jobs.

```
$ scons -j 10 test-cases-apps
```

(if user selects different config – it needs to be passed to that command too).

Tests themselves can be executed in a parallel fashion (`-j` option other than `1`).

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 16.

Listing 16 – Using `tee` to observe and store build logs at the same time.

```
$ scons  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 ...
Mkdir("build/release")
scons: done reading SConscript files.

config: Test Environment configuration file. ( /path/to/config )
    default: configs/default_sdram.conf
    actual: configs/default_sdram.conf

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

remoteJobTimeout: Timeout in seconds for remote job to end (GDB, log, etc.)
    default: 900
    actual: 900


CTSSW - CANopen SW Library Test Suite - v3.1.0
Licensed under European Space Agency Public License (ESA-PL) Permissive – v2.3
Copyright N7 Space sp. z o.o. 2020-2021


Use scons -H for help about 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 2 for details.

Table 2 – CTSSW SCons options.

Option	**Description**
CTSSW specific	
`config`	Path to configuration file, as described in 0. Detailed options available in Table 3.
`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.
`remoteJobTimeout`	Timeout for each operation using HWTB. Might need to be extended in case of some poor network performance.
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 3 – CTSSW configuration file options.

Option	**Description**
`[gdb]`	
`address`	Address of GDB Server to connect to.
`path`	Path to GDB executable.
`verbose`	`True/False` – When set, GDB will include additional information in SCons log (recommended).
`[gdbServer]`	
`address`	Address of the proxy to run GDB Server on, or `localhost` to run locally.
`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).
`[ioHandlerType]`	
`ioHandlerType`	Selection of the way tests applications should handle their standard output: • `sdram` – Standard output stored in SDRAM memory region, options available in `sdramIoHandler` section, • `uart` – Standard output sent using UART connected to USB dongle, options available in `uartIoHandler` section,

Option	Description
	• `usb` – Standard output sent using UART over USB HWTB interface, options available in `uartIoHandler` section. During CANSW validation the SDRAM option is used by default, to limit necessary links to the HWTB. It's biggest disadvantage is lack of live feedback from the application. UART based solutions are recommended for debugging.
`[sdramIoHandler]`	
`address`	Address of the proxy to run JLink server, or `localhost` to run locally.
`username`	Credentials required to access proxy via SSH.
`password`	
`path`	Path on the proxy (or local) to JLink executable.
`device`	JLink executable arguments (used to access memory).
`speed`	
`interface`	
`dataAddress`	Address of the memory buffer to be used for standard output.
`[uartIoHandler]`	
`address`	Address of the proxy to access connected UART device / dongle, or `localhost` to access locally.
`username`	Credentials required to access proxy via SSH.
`password`	
`baudrate`	Baudrate of the UART link.
`port`	Port to be used by `socat` to make UART available over Ethernet. Used even when setup locally.
`path`	Path to UART device on the proxy (or local) – e.g `/dev/ttyACM0`
`verbose`	`True/False` – When set, UART output is directly visible in the SCons log, before being saved to the log file.
`uartId`	UART device id on HWTB (`uart` mode only).
`[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).
`socatPort`	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).

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 17. Then the newly created folder needs its own SCons configuration file – `SConscript`, see Listing 18 for example. It contents will be filled in the next stages of this tutorial.

Listing 17 – 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 18 – 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 19.

Listing 19 – Example of C source file of new tests.

```
#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 20.

Listing 20 – 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 NAME-OF-THE-TEST
```

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

12 Analytical Index

N/A

13 Lists

13.1 List of Tables

13.2 List of Figures

13.3 List of Listings