Razorcat Technical Report

Date: 22 December 2010
Status Released

CCDL User Manual

Project	CCDL
Customer	Airbus

Summary:

This document describes the CCDL language structure and the statements available for writing CCDL test procedures. It also describes the process of writing CCDL user functions.

Reference/Related Documents:

User Manual for TRM clients

Notes:

Keywords	CCDL, TOP, TRM

Revision History

Revision Comment	Name	Date	Issue
Initial revision	Michael Wittner	06. July 2010	01
-	Michael Wittner	21. July 2010	02
-	Michael Wittner	8. September 2010	03
User functions handling updated, CCDL configuration added.	Michael Wittner	04. November 2010	04
Macro description added.	Michael Wittner	22. December 2010	05

Table of Contents

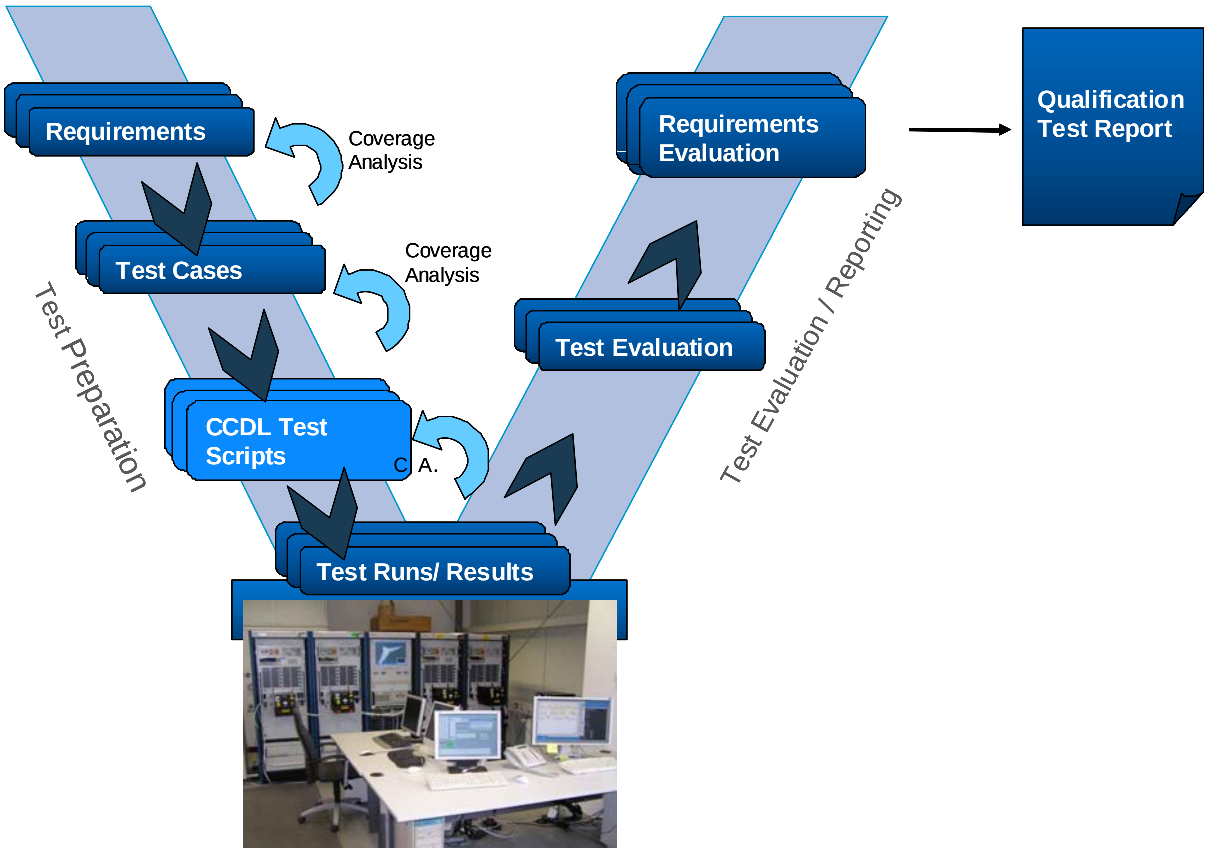
1 Introduction

Verification of safety critical systems requires full coverage of system under tests requirements. This results in many and complex test scenarios, to be executed and evaluated. Manual execution of such tests is error prone and not efficiently, though automated testing of the system under test (SUT) is required.

To improve the test coverage while using less human resources, there is a need for a tool, which allows to define test scenarios including the expected system reactions in a simple and unambiguous way, automatically run the test scenarios, automatically evaluate and report the behavior of the system under test after each test run.

The **c**heck **c**ase **d**efinition **l**anguage (CCDL) is an approach to automate system level testing by providing a high level script language that allows defining test stimulations and expected results in a human readable form. The CCDL bridges the gap between a purely textual description of a test and the compilation into a test stimulation program required by any automated test execution tool. A well defined interface to the underlying test execution engine allows execution of CCDL written tests on any test tool that provides the required functionality.

Moreover, CCDL is embedded into a complete testing process starting from the definition of tests, linking tests to system requirements, executing tests and review as well as reporting of test results as shown in the figure below (the V model development process).

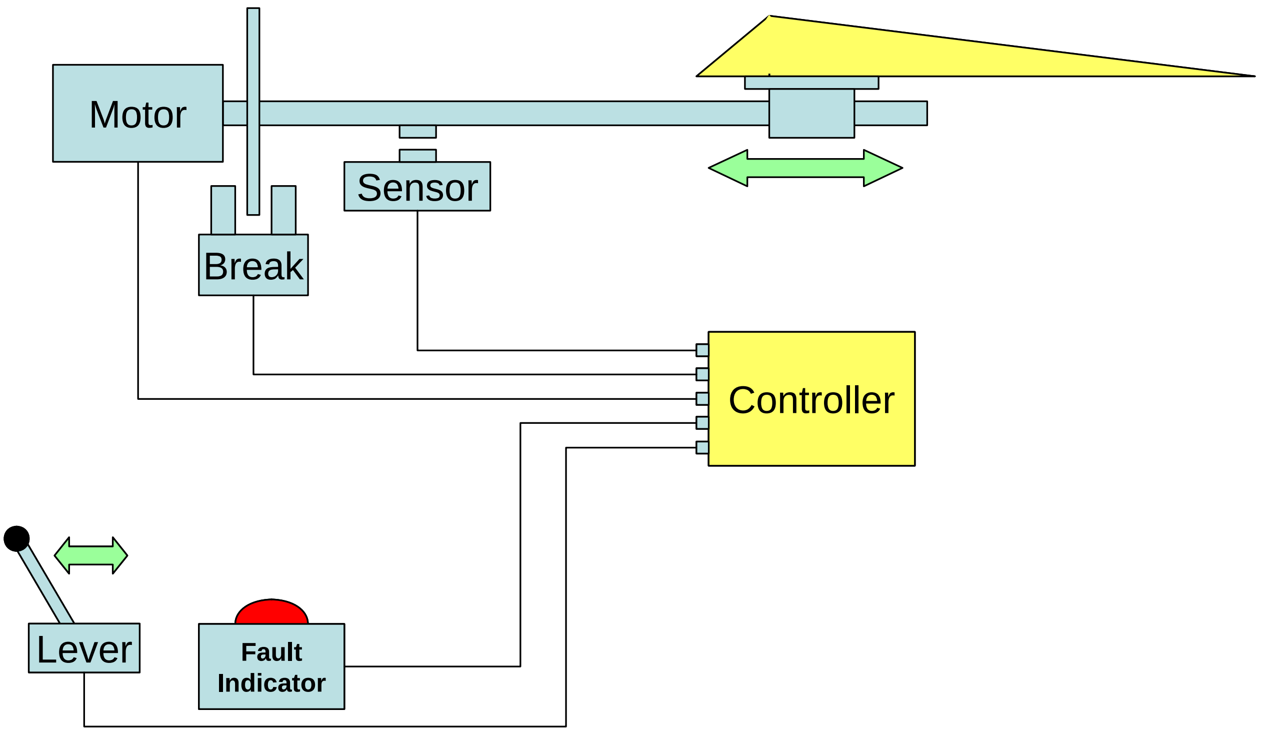


The CCDL testing process provides open interfaces to test management solutions and it is already integrated into the **I**ntegrated **T**est **E**nvironment (ITE) from Razorcat Development GmbH which supports the whole testing life cycle according to the V model mentioned above.

The CCDL language provides means to link individual expected reactions of the system under test to the respective system requirements. Such traceability of test results to system requirements and vice versa is one of the most important issues arising while testing safety critical systems according to aerospace, automotive or medical standards.

1 CCDL Sample

The following very simple actuator system of an airplane wing part shall illustrate the functionality of the CCDL. The system consists of a controller that controls the movements of a wing part depending on the lever setting (i.e. the lever is the input from the operator). The motor drives the wing part and the sensor measures speed and position of the system. The controller will be the SUT in the following example.



The system shall be verified against the requirements given within the specification of the system. The default position of the lever is 0 and it may be moved to positions 1 and 2. This drives the motor until the wing part comes into the respective position.

1.1 Requirements of the Sample System

As an excerpt from the system specification, the following requirements for the controller were selected and they shall be verified by means of system testing:

- RQMT:0815-1 The motor shall operate the system at a speed of 1000 rpm
- RQMT:4711-1 If any overspeed (more than 1100 rpm) is detected, the system shall stop the motor and activate the break within 100 ms. A fault warning shall be indicated.

1.2 Definition of Tests

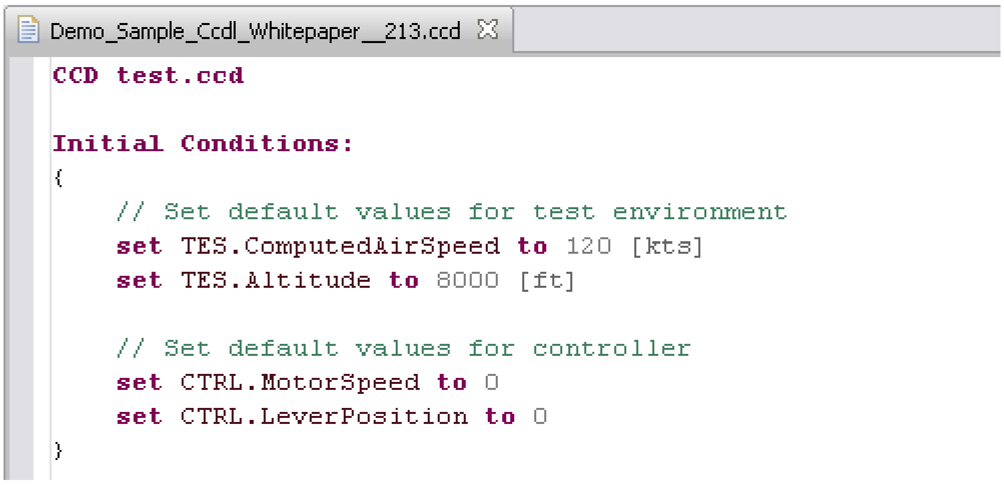
The next step in testing is the definition of test scenarios for the SUT. We will consider the following test definition for the overspeed tests:

- Reset the system to initial state and positions
- Set the lever position to position 1
- Wait until the motor has reached the normal speed (refer to requirement RQMT:0815-1)
- Simulate a sensor failure: Set the sensor to an offset of 110 rpm above the originally measured value (refer to requirement RQMT:4711-1)
- Check that the system gets stopped after 100 ms (refer to requirement RQMT:4711-1)

This test describes the steps to be taken in order to prepare the SUT for the test as well as the stimulation, error injection and the expected reaction of the SUT. The CCDL script will implement this test and provide means to automatically check the expected system reactions.

1.3 Initial Conditions of the Test

One of the prerequisites for the test are the initial conditions and settings of the SUT as well as the test bench. The CCDL provides the **Initial Condition** block to specify this initial setup for the test:



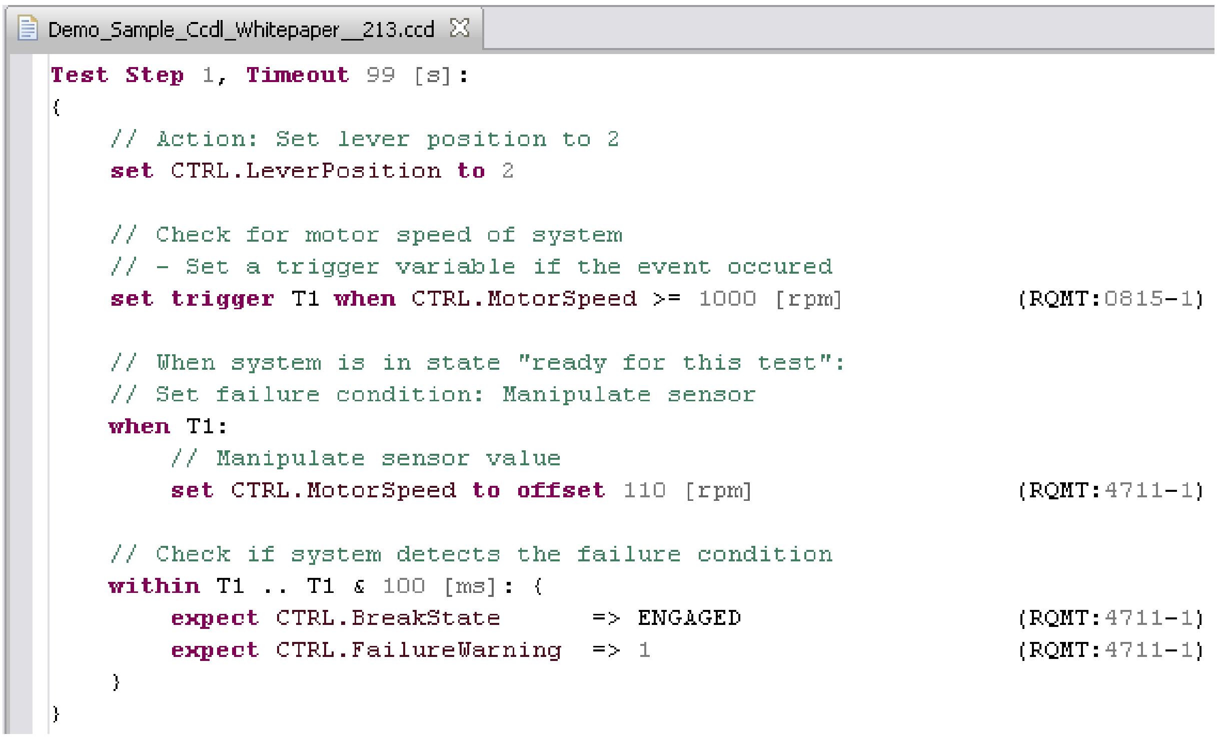
Demo_Sample_Ccdl_Whitepaper__213.ccd

```
CCD test.ccd

Initial Conditions:
{
    // Set default values for test environment
    set TES.ComputedAirSpeed to 120 [kts]
    set TES.Altitude to 8000 [ft]

    // Set default values for controller
    set CTRL.MotorSpeed to 0
    set CTRL.LeverPosition to 0
}
```

The controller is specified as **CTRL** whereas the test bench environment model is specified as **TES**. Parameters of both systems are initialized within the initial conditions block.

1.4 Test Steps

The stimulation of the test and the check for expected system reactions is carried out within test steps. The Test definition above may be tested with the CCDL implementation shown below


Demo_Sample_Ccdl_Whitepaper__213.ccd

```
Test Step 1, Timeout 99 [s]:
{
    // Action: Set lever position to 2
    set CTRL.LeverPosition to 2

    // Check for motor speed of system
    // - Set a trigger variable if the event occured
    set trigger T1 when CTRL.MotorSpeed >= 1000 [rpm]               (RQMT:0815-1)

    // When system is in state "ready for this test":
    // Set failure condition: Manipulate sensor
    when T1:
        // Manipulate sensor value
        set CTRL.MotorSpeed to offset 110 [rpm]                    (RQMT:4711-1)

    // Check if system detects the failure condition
    within T1 .. T1 & 100 [ms]: {
        expect CTRL.BreakState      => ENGAGED                     (RQMT:4711-1)
        expect CTRL.FailureWarning  => 1                           (RQMT:4711-1)
    }
}
```

This test step stimulates the system, waits for the system to operate properly, then injects the failure condition and finally checks for the expected reactions of the SUT.

This simple example already outlines the powerful language features of CCDL: The **trigger** expression denotes a certain point in time where the respective condition is fulfilled. Based on this trigger, the stimulation (the **when** statement) and expected reaction checks (the **within** statement) will be carried out at point in time where the SUT is in the desired state for testing. Time intervals (**T1 .. T1 & 100 [ms]**) using trigger expressions and offsets allow precise expected reaction checks in real time. The expected reaction operator **=>** is applicable for boolean expressions. It checks whether the value changes exactly once from the negated boolean value to the boolean value specified in the expression (within the given time interval).

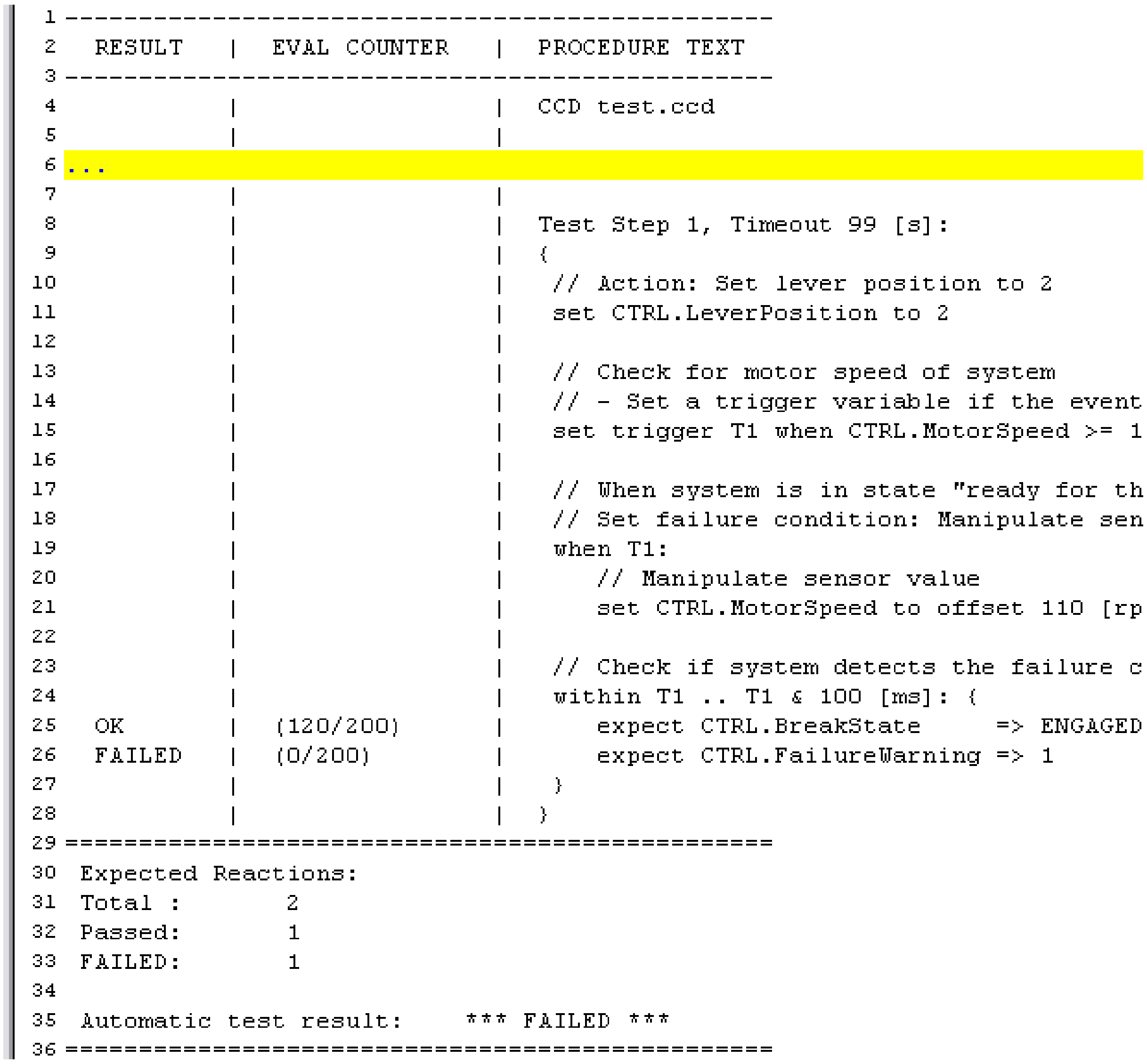
1.5 Test Preparation

Before executing the test, the CCDL script has to be compiled into an executable application that shall run on the test bench. The CCDL compiler produces C-Code that is executable on the test bench (through the virtual machine and based on the adaptable interface library). It may be integrated into the normal compilation process of the test bench.

1.6 Test Execution Result

During execution of the test, the initial condition settings will be applied and all specified test steps will be executed one after another. Test steps have an optional timeout period which will abort the test if the execution time exceeds the specified time. On successful test completion, the CCDL real time code generates an automatic evaluation result log file. This log file contains the procedure text and the passed/failed results of all expected reactions specified within the CCDL procedure.

Below is an excerpt of the result log file for the sample CCDL.



```
-----------------------------------------------
  RESULT    |  EVAL COUNTER    |  PROCEDURE TEXT
-----------------------------------------------
            |                  |  CCD test.ccd
            |                  |
...
            |                  |
            |                  |  Test Step 1, Timeout 99 [s]:
            |                  |  {
            |                  |   // Action: Set lever position to 2
            |                  |   set CTRL.LeverPosition to 2
            |                  |
            |                  |   // Check for motor speed of system
            |                  |   // - Set a trigger variable if the event
            |                  |   set trigger T1 when CTRL.MotorSpeed >= 1
            |                  |
            |                  |   // When system is in state "ready for th
            |                  |   // Set failure condition: Manipulate sen
            |                  |   when T1:
            |                  |      // Manipulate sensor value
            |                  |      set CTRL.MotorSpeed to offset 110 [rp
            |                  |
            |                  |   // Check if system detects the failure c
            |                  |   within T1 .. T1 & 100 [ms]: {
  OK        |  (120/200)       |      expect CTRL.BreakState      => ENGAGED
  FAILED    |  (0/200)         |      expect CTRL.FailureWarning => 1
            |                  |   }
            |                  |  }
===============================================
 Expected Reactions:
 Total :        2
 Passed:        1
 FAILED:        1

 Automatic test result:    *** FAILED ***
===============================================
```

2 CCDL Language

The CCDL syntax description will use the following notation:

- Expressions like <something> mean that this token will be replaced by a concrete value when applying the syntax rule.
- Syntax elements enclosed in [<any number of tokens>] are optional and may be omitted.

2.1 Case Sensitivity

All statements within a CCDL test procedure may be written in one of the following upper-/lowercase combination:

- All uppercase
- All lowercase
- First character in uppercase, the rest in lowercase

The following statements are all equivalent:

```
set HLSF1.FPPU to 300 [deg]
SET Hlsf1.fppu TO 300 [deg]
Set HLSF1.FPPU To 300 [deg]
```

It is recommended to write the statements in lowercase for better readability.

2.2 Comments

The CCDL supports C-style as well as C++-style comments. Comments may **not** be nested.

Sample comments:

```
// this is a C++ style comment
/* this is a C style comment
   over several lines */
```

The CCDL editor within TOP provides easy comment/uncomment actions. You may type **Ctrl-/** (i.e. **Ctrl-Shift-7** on a German keyboard) to comment or uncomment the currently selected line. If there are multiple lines selected, this action will comment all these lines. Invoking this action again will uncomment the selection again.

2.3 Structure of a CCDL Test Procedure

The CCDL test procedure consists of one block with initial conditions for the test and an arbitrary number of test steps that contain the test procedure actions and expected reactions. The following structure applies to CCDL test procedures:

```
CCD
<initial condition block>
<one or more test step blocks>
END OF CCD
```

The following shows an empty sample test procedure:

```
CCD

Initial Conditions:
{
}

Test Step 1:
{
}

END OF CCD
```

2.3.1 Initial Conditions

There must be exactly one initial conditions block at the beginning of the test procedure. The initial conditions block may contain any number of CCDL statements.

Normally, the initial conditions block will be used to setup the test preconditions and do other test preparations like starting the data logger or specifying monitoring functions that shall run for the whole duration of the test procedure. The following syntax applies to initial conditions:

```
Initial Conditions:
{
<any number of CCDL statements>
}
```

The following is a sample initial conditions block:

```
Initial Conditions:
{
      HLSVE.datalogger(START)
      wait 2 [s]
}
```

2.3.2 Test Steps

The rest of the CCDL test procedure consists of test step blocks that may contain arbitrary CCDL statements. The following syntax applies to test steps:

```
Test Step <number> [, Timeout <duration>]:
{
<any number of CCDL statements>
}
```

The test step numbers within the test procedure must be sequential (i.e. the first test step starts with “1” and all following test steps numbers must be incremented by one). The timeout specification is optional. If no timeout is given, the test step may run infinitely (depending on the contained commands). The following is a sample test step block:

```
Test Step 1, Timeout 99 [s]:
{
      set HLSF1.FPPU to 300 [deg]

      wait 5 [s]
}
```

2.3.3 Timeouts

As a test step option, you may specify a timeout for each test step. If the test step is not finished until the timeout time has elapsed, the whole test run will be aborted and an error message will be logged within the CCDL result.

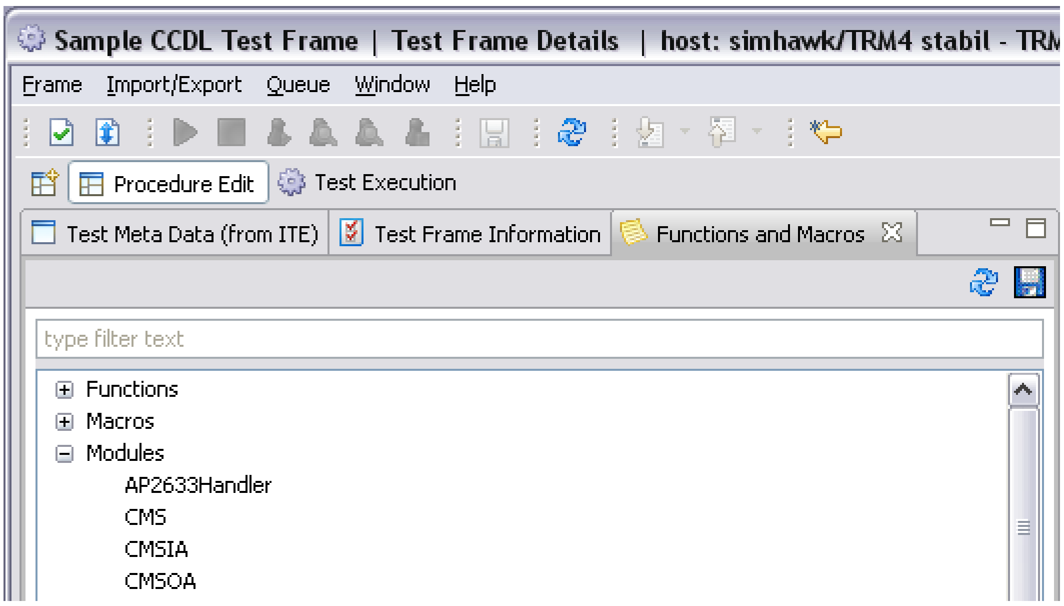
2.3.4 Modules

Modules are the functional units of either the UUT or the system environment. Every parameter and every user function are related to a module. The first part of every parameter or user function name is the module name followed by a dot and the name of the parameter or user function itself. The following gives an example of a parameter name and a user function:

```
      set HLSS1.SLAT_FPPU_RH to 300 [deg]
      HLSVE.datalogger(START)
```

The module **HLSS1** is part of the UUT functional units whereas the module **HLSVE** is part of the test environment.

The list of available modules is shown within the **Functions and Macros** view of TOP (in the procedure edit perspective).


Sample CCDL Test Frame | Test Frame Details | host: simhawk/TRM4 stabil - TRM
Frame Import/Export Queue Window Help
Procedure Edit
Test Execution
Test Meta Data (from ITE)
Test Frame Information
Functions and Macros
type filter text
Functions
Macros
Modules
AP2633Handler
CMS
CMSIA
CMSOA

2.3.5 Parameters

The parameters are the stimulation and measurement interface for the UUT and the test environment. Parameter values may be stimulated and expected values may be specified using CCDL statements. The parameter name consists of the module name where the respective parameter is related to and the parameter name itself. Both name components are separated by a dot.

```
set HLSS1.SLAT_FPPU_RH to 300 [deg]
```

All available parameters are shown within the **Parameter Browser** view of TOP. They are also available within the auto-completion popup menu of the TOP editor itself. Please refer to the TOP user manual for details.

2.3.6 Multi Parameter Specification

The CCDL provides means to specify multiple parameters at once within a statement. Such a multi parameter statement will be handled in the same way, as if the statement would have been written for each parameter.

```
set HLSS[1;2].SLAT_FPPU_[R;L]H to 300 [deg]
```

The following statements are identical to the above statement (with respect to functional aspects):

```
set HLSS1.SLAT_FPPU_RH to 300 [deg]
set HLSS1.SLAT_FPPU_LH to 300 [deg]
set HLSS2.SLAT_FPPU_RH to 300 [deg]
set HLSS2.SLAT_FPPU_LH to 300 [deg]
```

2.3.7 Constants

Constants may be used within the CCDL test procedure at every location where a number may be specified. The constant will simply be replaced by the corresponding value during compilation of the test procedure.

The available constants are visible within the **Functions and Macros** view of TOP. They are also available within the auto-completion popup menu of the TOP editor itself. Please refer to the TOP user manual for details.

2.3.8 Units

The CCDL language supports units for calculation and assignment of values to parameters. Units are written in "[ ]" after a value specification like follows:

```
set HLSS1.SLAT_FPPU_RH to 300 [deg]
```

Each parameter may have a unit assigned. In this case, any assignment of values will be converted automatically into the unit of the parameter. If the unit given with the assigned value and the parameter unit does not match, the compiler will try to convert the value into the unit of the parameter. If this is not possible, the compiler will issue an error message.

The following three value assignments will set the same value (20 [deg]) for the given parameter:

```
set HLSS1.SLAT_FPPU_RH to 20
set HLSS1.SLAT_FPPU_RH to 20 [deg]
set HLSS1.SLAT_FPPU_RH to 0.3 [rad]
```

2.3.9 Expressions

Expressions may be used within statements of the CCDL language. The simplest expression is just a number or a parameter. The following section lists the arithmetical and boolean operators that are available to form expressions.

2.3.9.1 Arithmetical Expressions

The following arithmetical expressions are available:

expression1	**+**	*expression2*	=> Addition
expression1	**-**	*expression2*	=> Subtraction
expression1	*****	*expression2*	=> Multiplication
expression1	**/**	*expression2*	=> Division
expression1	**%**	*expression2*	=> Modulo operator
expression1	**^**	*expression2*	=> expression1 *power of* expression2

abs	**(**	*expression*	**)**	=> Absolute value
sin	**(**	*expression*	**)**	=> Sine
cos	**(**	*expression*	**)**	=> Cosine
tan	**(**	*expression*	**)**	=> Tangens
asin	**(**	*expression*	**)**	=> Arcussinus
acos	**(**	*expression*	**)**	=> Arcuscosinus
atan	**(**	*expression*	**)**	=> Arcustangens
sinh	**(**	*expression*	**)**	=> Sinushyperbolikus

cosh	**(**	*expression*	**)**	=> Cosinushyperbolikus
tanh	**(**	*expression*	**)**	=> Tangenshyperbolikus
exp	**(**	*expression*	**)**	=> Exponentialfunktion
log	**(**	*expression*	**)**	=> Logarithmus
log10	**(**	*expression*	**)**	=> Logarithmus zur Basis 10
sqrt	**(**	*expression*	**)**	=> Square root

The normal arithmetic rules for operator precedence and usage of parentheses apply. Multi selection of parameter names are not allowed within arithmetic expressions.

2.3.9.2 Logical Expressions

Logical expressions are all expressions that resolve to a boolean value. They either have the value true (1) or false (0).

The following logical operators are available within CCDL:

expression1	**<**	*expression2*	true, if expression1 less than expression2
expression1	**<=**	*expression2*	true, if expression1 less or equal than expression2
expression1	**=**	*expression2*	true, if expression1 equal to expression2
expression1	**>=**	*expression2*	true, if expression1 greater or equal than expression2
expression1	**>**	*expression2*	true, if expression1 greater than expression2
expression1	**and**	*expression2*	true, if expression1 and expression2 are true
expression1	**or**	*expression2*	true, if expression1 or expression2 is true
	not	*expression1*	true, if expression1 is false

expression1 **between** *expression2* **and** *expression3*
true, if expression1 resolves to a value within the range given by expression2 and expression3

The expressions may contain unit specifications if applicable. For the **between** operator, only the expression2 and expression3 may contain unit specifications, if expression1 is a parameter that has a unit assigned.

The following example illustrates the usage of logical expressions:

```
HLSS1.SLAT_FPPU_RH <= 0.3 [rad]
```

The usage of the unit specification is correct, because the parameter HLSS1.SLAT_FPPU_RH has the unit [deg], which may be converted into [rad].

The expression resolves to **true** as long as HLSS1.SLAT_FPPU_RH is less than or equal to 0.3 [rad].

When HLSS1.SLAT_FPPU_RH is greater than this value (e.g. 0.31 [rad]), the expression will resolve to **false**.

2.3.9.3 Multi Parameter Specification in Logical Expressions

Within logical expressions, it is possible to specify multi parameter names. The resulting expression will be build by filling the logical expression with each of the parameters from the resulting multi parameter list and combining them with the logical **and** operator.

The following example will be expanded internally to the expression shown below:

```
HLSS[1;2].SLAT_FPPU_RH <= 30
```

This will result in the following expression:

```
(HLSS1.SLAT_FPPU_RH <= 30) and (HLSS2.SLAT_FPPU_RH <= 30)
```

2.3.10 Variables

The CCDL language also provides variables within the test procedure. They may be declared and initialized like follows:

```
set variable sample_var to 300 [deg]
```

Such variables may be used within any logical or arithmetic expression after the position of the variable declaration within the test procedure like follows:

```
when HLSS1.SLAT_FPPU_RH > sample_var:
      expect …

expect sample_var < 200
```

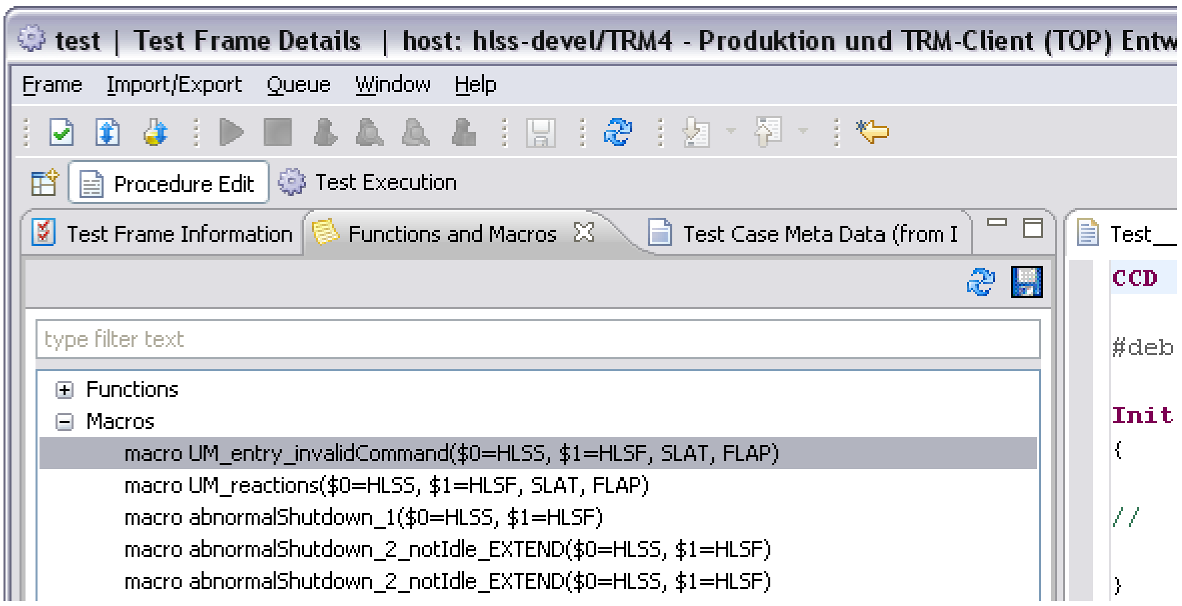
If already declared and used variables shall be updated with a new value, you need to use the same syntax as for the declaration/initialization:

```
set variable sample_var to 299 [deg]
```

2.3.11 Macros

The CCDL language also provides macros for easy reuse of code fragments. If you write a piece of CCDL code which you want to reuse or share with other team members, you may create a macro and insert this code fragment into the macro body. Macros may have parameters in order to conditionally show or hide fragments of the macro code and to propagate values into the generated macro code when the macro gets expanded.

The list of available macros will be shown within the **Functions and Macros** view of TOP. The source for the macros list is the **macros.ccd** file within the configuration directory of the CCDL compiler (Refer to chapter 4).


test | Test Frame Details | host: hlss-devel/TRM4 - Produktion und TRM-Client (TOP) Entw
Frame Import/Export Queue Window Help
Procedure Edit
Test Execution
Test Frame Information
Functions and Macros
Test Case Meta Data (from I
Test__
type filter text
Functions
Macros

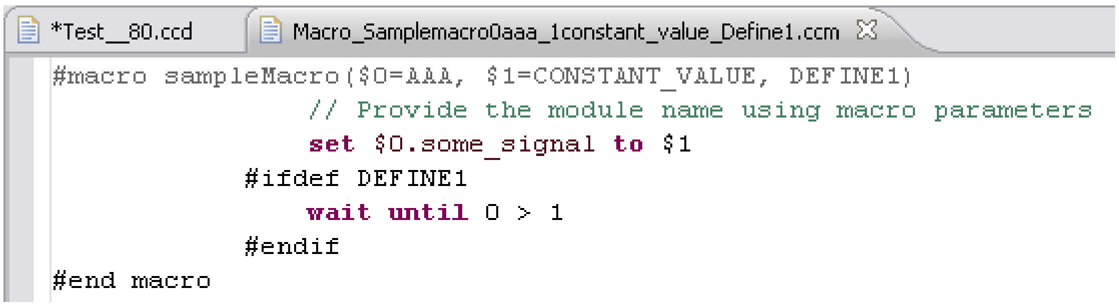
```
macro UM_entry_invalidCommand($0=HLSS, $1=HLSF, SLAT, FLAP)
macro UM_reactions($0=HLSS, $1=HLSF, SLAT, FLAP)
macro abnormalShutdown_1($0=HLSS, $1=HLSF)
macro abnormalShutdown_2_notIdle_EXTEND($0=HLSS, $1=HLSF)
macro abnormalShutdown_2_notIdle_EXTEND($0=HLSS, $1=HLSF)
```

Macro headers may be inserted into the test procedure code using drag&drop. If the list of macros is empty, at least one macro need to be inserted into the macros.ccd configuration file. You may also edit the macros.ccd file directly using any external editor.

2.3.11.1 Editing Macros

Double-clicking on a macro opens the macro editor. Each macro definition is enclosed using the **#macro** and **#end macro** keywords like shown below.

You may create a new macro either by adding the new macro definition before the currently displayed macro within the macro editor or by copying, pasting and renaming any existing macro within the macro editor.



```
#macro sampleMacro($0=AAA, $1=CONSTANT_VALUE, DEFINE1)
                 // Provide the module name using macro parameters
                 set $0.some_signal to $1
             #ifdef DEFINE1
                 wait until 0 > 1
             #endif
#end macro
```

A macro definition consists of the following parts:

- Macro name
- Parameter list with numbered entries starting from $0 up to $9 (this is the maximum number of parameters). Each of these parameters needs to have an identifier value assigned.
- Additional entries within the parameter list that will be used as defines within the macro body
- The macro body containing the CCDL code of the macro

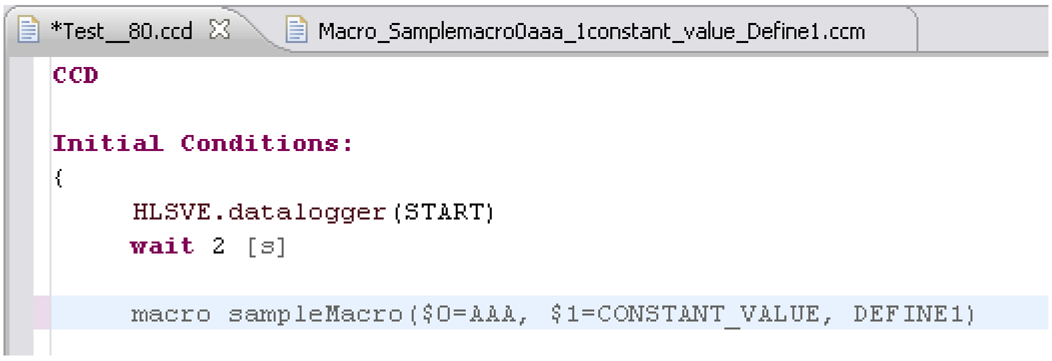
Within the macro body, code fragments may be conditionally enabled using the **#ifdef** and **#endif** keywords. If the respective define is provided within the parameter list, the corresponding code fragment will be used when expanding the macro.

Pressing the **Save** button will save the new macro definition or any changes of existing macros to the macros file.

The **Refresh** button will trigger reading the macros file again and update the list of macros displayed within the **Functions and Macros** view.

2.3.11.2 Expanding Macros

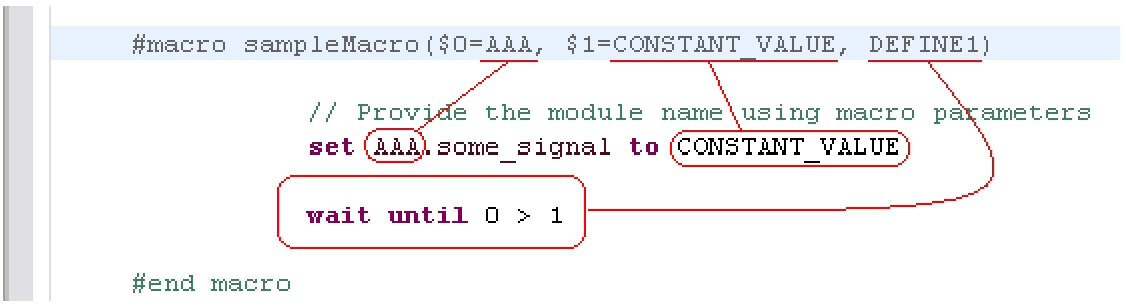
Macro headers may be inserted into the test procedure code using drag&drop from the **Functions and Macros** view. This will only add the header of the macro beginning with the **macro** keyword (this indicates that the macro has not been expanded yet).



```
CCD

Initial Conditions:
{
      HLSVE.datalogger(START)
      wait 2 [s]

      macro sampleMacro($0=AAA, $1=CONSTANT_VALUE, DEFINE1)
```

When pressing the **Expand Macros** button within the toolbar, all macros within the current test procedure will be expanded (You need to save the procedure before starting the macro expansion).

The expanded procedure will contain the macro code with the macro parameters (e.g. $1) replaced by their respective parameter values. Also any defines specified within the parameter list will control if parts of the macro code enclosed with #ifdef and #endif will be expanded.



```
#macro sampleMacro($0=AAA, $1=CONSTANT_VALUE, DEFINE1)

            // Provide the module name using macro parameters
            set AAA.some_signal to CONSTANT_VALUE

            wait until 0 > 1

#end macro
```

The expanded macro definition will be enclosed with the **#macro** and **#end macro** keywords. Another expansion of already expanded macros will replace the current macro body (within the test procedure) with the latest contents of the macro definition.

2.4 CCDL Statements

2.4.1 Value Manipulation Statements

All manipulator statements will manipulate the value of a parameter until the parameter is released again using the **release** statement. In case of using the **offset** option, the manipulator statements will continuously read the original value of the parameter, add the given offset and write this manipulated value to the parameter.

2.4.1.1 set

Syntax:

set <identifier> to [offset] <expression> [<unit>]
[for <expression> <time unit>]
[until <logical expression>]

Short form: **<identifier> := <expression> [<unit>]**

This statement sets the parameter with the given identifier (or list of parameters in case of a multi parameter identifier) to the given value. The following options are available for this statement:

- The **for** option manipulates the parameter value for the specified duration time.
- The **until** option manipulates the parameter value until the specified logical expression is true.

The **offset** option manipulates the parameter continuously with the given offset until the parameter is released again using the **release** statement.

2.4.1.2 hold/freeze

Syntax:

(hold | freeze) <identifier>
[for <expression> <time unit>]
[until <logical expression>]

This statement freezes the parameter with the given identifier (or list of parameters in case of a multi parameter identifier) to the current value. Both statements (hold/freeze) are semantically equivalent. The following options are available for this statement:

- The **for** option freezes the value for the specified duration time.
- The **until** option freezes the value until the specified logical expression is true.

The **hold/freeze** statement freezes the parameter until the parameter is released again using the **release** statement.

2.4.1.3 ramp

Syntax:

ramp <identifier> with rate <constant> [<unit> "/" <time unit>]
[from <constant> [<unit>]]
[to [offset] <constant> [<unit>]]
[for <expression> <time unit>]
[until < logical expression >]

This statement manipulates the parameter with the given identifier (or list of parameters in case of a multi parameter identifier) with a series of values that represent a ramp function with the given ramp rate. The following options are available for this statement:

- The **from** option creates a ramp starting at the given value (regardless of the current parameter value; this may cause a value jump from the current value to the value given with the **from** option).
- The **to** option creates a ramp to the given value. The ramp function will terminate immediately, if the current parameter already has this value.
- The **for** option manipulates the parameter for the specified duration time.
- The **until** option manipulates the parameter until the specified logical expression is true.

The **offset** option manipulates the parameter continuously with the given offset until the parameter is released again using the **release** statement.

2.4.1.4 increment/decrement

Syntax:

(increment | decrement) <identifier>
by <constant> [<unit>] each <constant> [<time unit>]
[from <constant> [<unit>]]
[to [offset] <constant> [<unit>]
| for <constant> <time unit>
| until < logical expression >]

This statement increments or decrements the parameter with the given identifier (or list of parameters in case of a multi parameter identifier). The following options are available for this statement:

- The **from** option increments/decrements starting at the given value (regardless of the current parameter value; this may cause a value jump from the current value to the value given with the **from** option).

- The **to** option increments/decrements to the given value. The statement will terminate immediately, if the current parameter already has this value.
- The **for** option increments/decrements the parameter for the specified duration time.
- The **until** option increments/decrements the parameter until the specified logical expression is true.

The **offset** option manipulates the parameter continuously with the given offset until the parameter is released again using the **release** statement.

2.4.1.5 release

Syntax: **release <identifier> [with rate <constant> [<unit> „/“ <time unit>]]**

This statement releases the parameter with the given identifier (or list of parameters in case of a multi parameter identifier) to the original value.

The option **with rate** is optional and you may specify a rate value that shall be used to release the parameter value stepwise until the original value will be reached.

2.4.2 Expected Reaction Statements

2.4.2.1 expect

Syntax: **expect <logical expression>**

Each expected reaction will be evaluated during the test execution. The result may either be passed or failed and it will be reported within the evaluation log file. The overall result of the test run is the sum of the individual expected reaction statement results. Any failed expected reaction will cause the overall test run result to be failed too.

2.4.2.2 check transitions of

Syntax: **check transitions of <identifier> = <expression>**

The value changes of the given identifier will be checked. If the number of value changes is the same as stated within the given expression, the statement result will be passed otherwise failed. The number of changes will be written into the evaluation result log.

Value changes may be checked during the whole test run as well as for the duration of test steps. The position of the "check transitions of" statement within the test procedure defines the time range for the check:

- If the statement is placed into the initial conditions block, the check will last for the rest of the whole test run.
- If the statement is placed into any test step block, the check will last for the rest of the duration of the respective test step.

2.4.2.3 monitor

Syntax: **monitor <logical expression>**

This statement checks whether the given logical expression is valid within a certain time range. Logical expressions may be monitored during the whole test run as well as for the duration of test steps. The position of the "monitoring" statement within the test procedure defines the time range for the check:

- If the statement is placed into the initial conditions block, the check will last for the rest of the whole test run.
- If the statement is placed into any test step block, the check will last for the rest of the duration of the respective test step.

2.4.3 Control Flow Statements

2.4.3.1 wait

Syntax: **wait <constant> <time unit>**

This statement waits for the specified time before executing the next statement within the test procedure.

2.4.3.2 wait until

Syntax: **wait until <logical expression> <time unit>**

This statement waits until the given logical expression is true before executing the next statement within the test procedure.

2.4.3.3 now

Syntax: **now: <user function call>**

This statement invokes the given user function within a special mode: The execution of the test procedure statements will continue regardless of the user function behaviour, i.e. if the user function will run for more than one execution cycle, the next statement(s) within the test procedure will be executed in parallel.

This is necessary for user functions that shall execute in parallel to the normal procedure execution control flow.

2.4.4 Trigger Statements

Trigger statements provide event based control of the test procedure execution either for stimulation or expected reaction evaluation. Each trigger block will be processed automatically upon activation of the trigger condition. This enables in-parallel execution of expected reaction checks during the normal test procedure control flow.

Each trigger block contains at least one statement. Multiple statements need to be enclosed into curly brackets.

2.4.4.1 Trigger Variables

Trigger conditions may be stored within trigger variables for further usage.

Syntax: **set trigger <identifier> when <logical expression>**
[& <constant> <time unit>]

These trigger variables may be used within expressions following the line of the trigger definition.

2.4.4.2 at/when/after

Syntax: (**at | when | after) <logical expression> : <trigger block>**

This statement causes all statements contained within the trigger block to be executed once in parallel as soon as the logical expression is true.

The **at**, **when** and **after** statements are semantically equivalent.

2.4.4.3 within

Syntax: **within <logical expression 1> [& <constant> <time unit>]**
.. < logical expression 2> [& <constant> <time unit>] :
<trigger block>

This statement causes all statements contained within the trigger block to be executed in parallel to the normal test procedure control flow for a certain time range. The start of the time range is defined by the first logical expression: As soon as this expression is true, the statements of the trigger block will be executed for each time frame until the second logical expression is true.

The optional **&** clause may be used to specify a time offset for the start and/or end of the time range. In such a case, the time range will start when the logical expression is true and the offset time has elapsed.

The **expect** statement has a special semantic when used within a trigger block of the **within** statement: The results of all (repeated) calls to **expect** statements during the invocation time range of the trigger block will be evaluated and summarized. If at least one of the (repeated) calls to an **expect** statement within the time range is true, the **expect** statement will be marked as passed within the evaluation result log.

2.4.4.4 during

Syntax: **during <logical expression 1> [& <constant> <time unit>]**

.. <logical expression 2> [& <constant> <time unit>] :

<trigger block>

This statement behaves mostly the same like the **within** statement, the difference is the handling of **expect** statements within the trigger block.

The **during** statement causes all statements contained within the trigger block to be executed in parallel to the normal test procedure control flow for a certain time range. The start of the time range is defined by the first logical expression: As soon as this expression is true, the statements of the trigger block will be executed for each time frame until the second logical expression is true.

The optional **&** clause may be used to specify a time offset for the start and/or end of the time range. In such a case, the time range will start when the logical expression is true and the offset time has elapsed.

The **expect** statement has a special semantic when used within a trigger block of the **during** statement: The results of all (repeated) calls to **expect** statements during the invocation time range of the trigger block will be evaluated and summarized. Only If all of the (repeated) calls to an **expect** statement within the time range are true, the **expect** statement will be marked as passed within the evaluation result log.

2.4.5 Control Functions for Pins

The CCDL language supports operations like short circuit or disconnect/reconnect on hardware connection pins of the UUT (i.e. the cable hardware which connects the UUT to the test bench). The test bench need to provide the means to electronically switch the cabling (e.g. with a relay matrix). The CCDL statements operate on pins which are provided within the configuration files.

2.4.5.1 short circuit

Syntax: **short circuit pin <pin name 1> and <pin name 2>**
[for <constant> <time unit>]

This statement connects both pins in order to create a short circuit between the pins of the physical I/O interface of the UUT. You need to release the short circuit later within your test procedure using the release short circuit statement.

You may optionally specify a duration time for the short circuit. In this case, both pins will be released automatically after the specified time.

2.4.5.2 release short circuit

Syntax: **release short circuit pin <pin name 1> and <pin name 2>**

This statement disconnects both pins in order to release a short circuit between the previously connected pins of the physical I/O interface of the UUT.

2.4.5.3 disconnect

Syntax: **disconnect pin <pin name 1> [, <pin name 2>]**
[for <constant> <time unit>]

This statement disconnects the given list of pins of the physical I/O interface of the UUT.

You may optionally specify a duration time for the disconnect operation. In this case, all pins will be reconnected automatically after the specified time.

2.4.5.4 reconnect

Syntax: **reconnect pin <pin name 1> [, <pin name 2>]**
[for <constant> <time unit>]

This statement disconnects both pins in order to release a short circuit between the previously connected pins of the physical I/O interface of the UUT.

You may optionally specify a duration time for the reconnect operation. In this case, all pins will be disconnected automatically after the specified time.

2.4.6 Other Statements

2.4.6.1 print

Syntax: **print <identifier>**

Prints the value of the given parameter into the log file.

2.4.7 User Functions

Syntax: **<user function identifier> ([<argument list>])**

Besides the built-in statements, any of the available user functions may be used within CCDL expressions. The invocation of a user function is declared by writing the function identifier followed by a list of arguments to the user function (this list may be empty). An example of a call to a user function is shown below:

```
HLSENV.selectFlapsLever(Position=1, duration=2[sec])
HLSENV.selectFlapsLever(1, 2[sec])

HLSVE.datalogger(1)
HLSVE.datalogger(START)
```

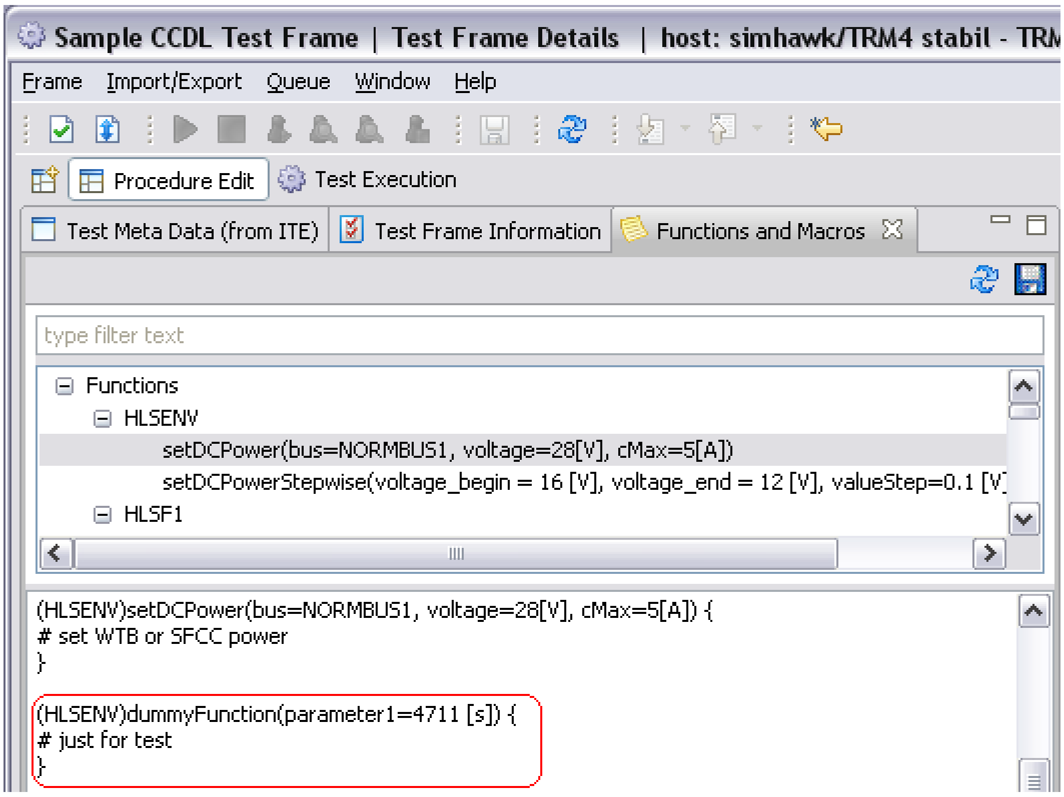
The above two sets of calls to user functions are equivalent, you may use the parameter name to provide a value or you may just write the argument list in the correct order like within the user functions definition.

3 CCDL User Functions

The CCDL language provides user functions in order to encapsulate test bench specific or complicated functionality into easy-to-handle CCDL language extensions. Each user function consists of a declaration within the configuration files of TOP and the corresponding implementation that needs to be compiled into the user function library.

3.1 Adding/Editing a User Function Declaration

The first step in adding a user function is to add the function declaration into the configuration files of TOP. Within the **Functions and Macros** view, select any user function and click into the text window that will appear below the user functions list:


Sample CCDL Test Frame | Test Frame Details | host: simhawk/TRM4 stabil - TRM
Frame Import/Export Queue Window Help
Procedure Edit
Test Execution
Test Meta Data (from ITE)
Test Frame Information
Functions and Macros
type filter text
Functions
HLSENV
HLSF1

```
setDCPower(bus=NORMBUS1, voltage=28[V], cMax=5[A])
setDCPowerStepwise(voltage_begin = 16 [V], voltage_end = 12 [V], valueStep=0.1 [V]
```

```
(HLSENV)setDCPower(bus=NORMBUS1, voltage=28[V], cMax=5[A]) {
# set WTB or SFCC power
}

(HLSENV)dummyFunction(parameter1=4711 [s]) {
# just for test
}
```

Add the declaration of the new user function. The argument list of the user function may contain arguments according to the following specification:

- Name of the argument
- Optional assignment of a default value
- Optionally followed by a unit specification

Press the **Save** button to save the user function declaration to the local configuration file. You need to check-in the files into the SVN server in order to propagate the new declaration to other users. Refer to the TOP user manual for details.

3.2 Implementing a User Function

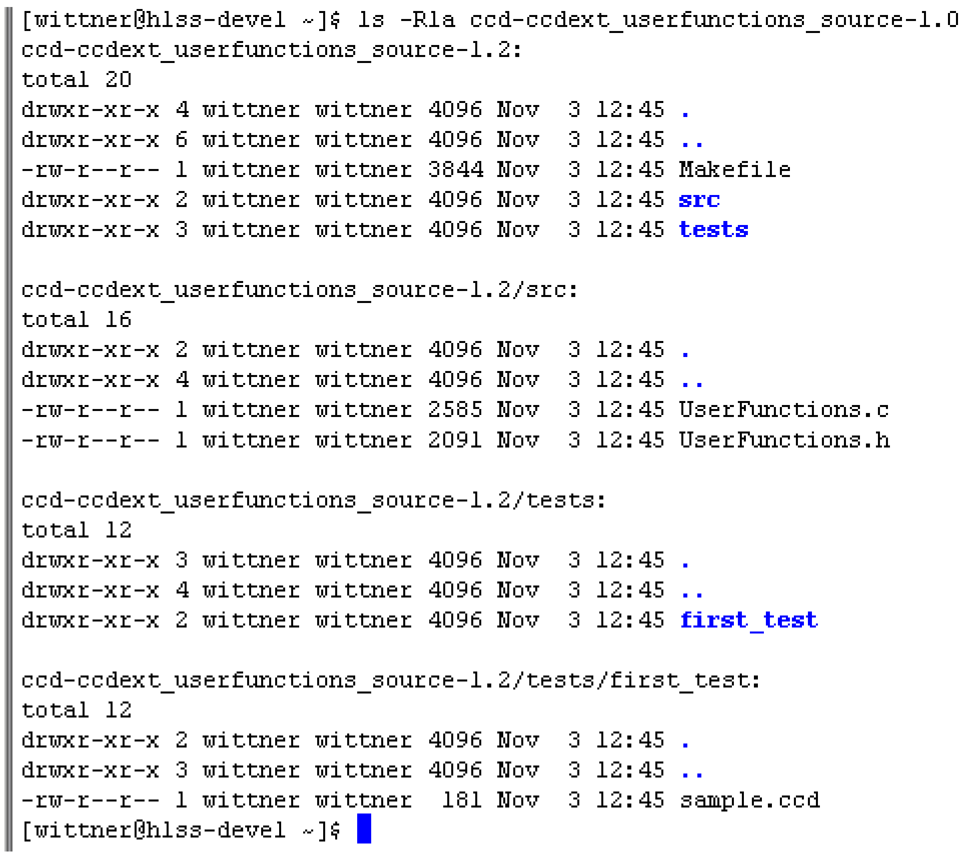
In order to implement a user function, you need to install the following RPM (with the newest available version number):

```
ccd-ccdext_userfunctions_source-1.0-1.i386.rpm
```

This will install the (initially almost empty) source files for the user function library that will be linked to the test runner binary. The package also includes a build environment for development and test of the user functions. After installation of the RPM, you need to **copy** the following directory somewhere into your private folder structure:

```
cp –R /usr/share/doc/ccd-ccdext_userfunctions_source-1.0 ~
```

This directory contains the following sub folders:



```
[wittner@hlss-devel ~]$ ls -Rla ccd-ccdext_userfunctions_source-1.0
ccd-ccdext_userfunctions_source-1.2:
total 20
drwxr-xr-x 4 wittner wittner 4096 Nov  3 12:45 .
drwxr-xr-x 6 wittner wittner 4096 Nov  3 12:45 ..
-rw-r--r-- 1 wittner wittner 3844 Nov  3 12:45 Makefile
drwxr-xr-x 2 wittner wittner 4096 Nov  3 12:45 src
drwxr-xr-x 3 wittner wittner 4096 Nov  3 12:45 tests

ccd-ccdext_userfunctions_source-1.2/src:
total 16
drwxr-xr-x 2 wittner wittner 4096 Nov  3 12:45 .
drwxr-xr-x 4 wittner wittner 4096 Nov  3 12:45 ..
-rw-r--r-- 1 wittner wittner 2585 Nov  3 12:45 UserFunctions.c
-rw-r--r-- 1 wittner wittner 2091 Nov  3 12:45 UserFunctions.h

ccd-ccdext_userfunctions_source-1.2/tests:
total 12
drwxr-xr-x 3 wittner wittner 4096 Nov  3 12:45 .
drwxr-xr-x 4 wittner wittner 4096 Nov  3 12:45 ..
drwxr-xr-x 2 wittner wittner 4096 Nov  3 12:45 first_test

ccd-ccdext_userfunctions_source-1.2/tests/first_test:
total 12
drwxr-xr-x 2 wittner wittner 4096 Nov  3 12:45 .
drwxr-xr-x 3 wittner wittner 4096 Nov  3 12:45 ..
-rw-r--r-- 1 wittner wittner  181 Nov  3 12:45 sample.ccd
[wittner@hlss-devel ~]$
```

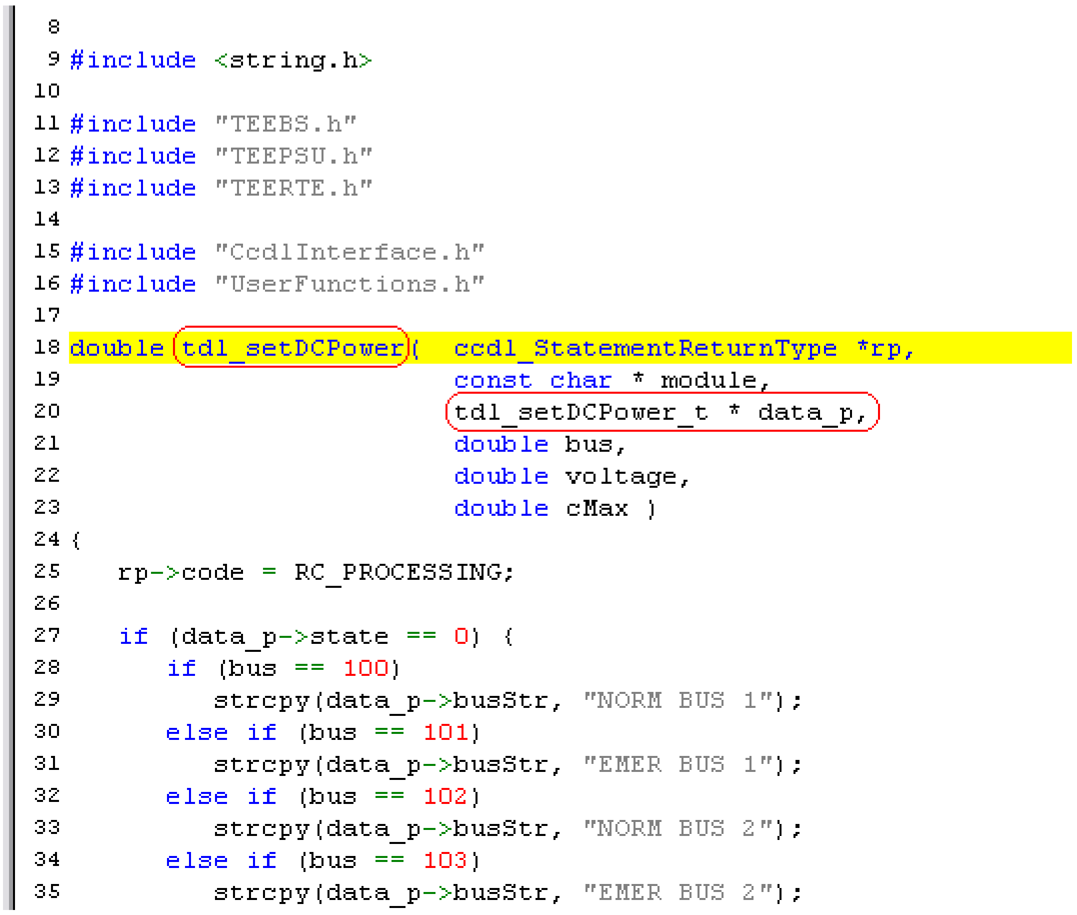
The **`UserFunctions.c`** file contains the source code for the implementation of the user function and the **`UserFunctions.h`** file contains the necessary declarations for the user function. As an example, we will have a look into the implementation of the **`tdl_setDCPower`** function that is already implemented.

The sub directory **`tests`** contains a sample CCDL procedure. This directory may contain any number of subdirectories with CCDL procedure files. All of them will be treated as separate tests and will be compiled and linked when invoking the build and test process.

Please note: In order to test for correct compiling and linking of your user function, you need to have at least one test with a CCDL procedure that really **uses** your user function.

3.2.1 Source File Contents

The source file (.c) contains the implementation of the user function.



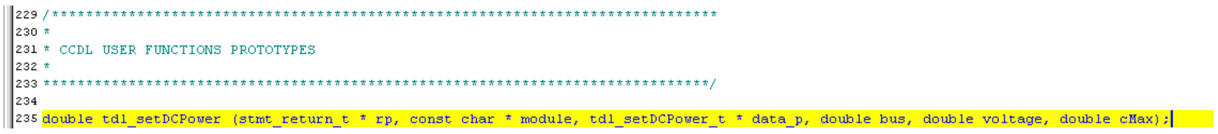
```

#include <string.h>

#include "TEEBS.h"
#include "TEEPSU.h"
#include "TEERTE.h"

#include "CcdlInterface.h"
#include "UserFunctions.h"

double tdl_setDCPower(  ccdl_StatementReturnType *rp,
                        const char * module,
                        tdl_setDCPower_t * data_p,
                        double bus,
                        double voltage,
                        double cMax )
{
   rp->code = RC_PROCESSING;

   if (data_p->state == 0) {
      if (bus == 100)
         strcpy(data_p->busStr, "NORM BUS 1");
      else if (bus == 101)
         strcpy(data_p->busStr, "EMER BUS 1");
      else if (bus == 102)
         strcpy(data_p->busStr, "NORM BUS 2");
      else if (bus == 103)
         strcpy(data_p->busStr, "EMER BUS 2");
```

The first three arguments are mandatory and the type of the third parameter is built from the name of the user function itself:

Name of user function	`xxx`
Type name of third argument	`xxx_t`

3.2.2 Header File Contents

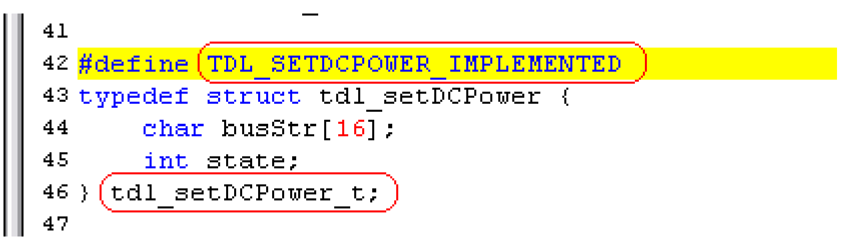
The header file (.h) contains the C-style declaration of the user function. Each user function needs to be declared as prototype according to the implementation:



```
/*****************************************************************************
*
* CCDL USER FUNCTIONS PROTOTYPES
*
*****************************************************************************/

double tdl_setDCPower (stmt_return_t * rp, const char * module, tdl_setDCPower_t * data_p, double bus, double voltage, double cMax);
```

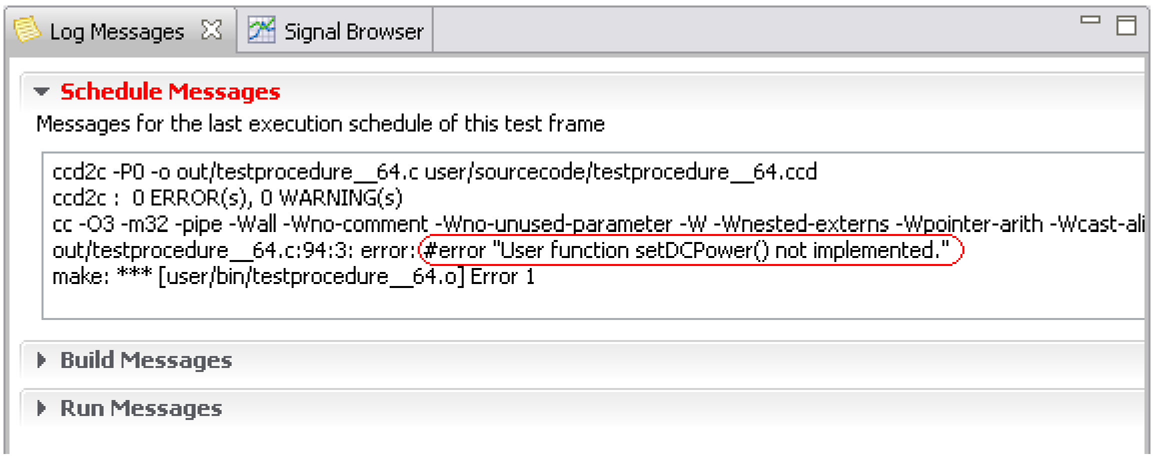
Additionally, there are the following declarations necessary:

- the type of the third argument needs to be declared
- a #define needs to be added in order to tell the system that this user function is ready to be used



```

#define TDL_SETDCPOWER_IMPLEMENTED
typedef struct tdl_setDCPower {
    char busStr[16];
    int state;
} tdl_setDCPower_t;

```

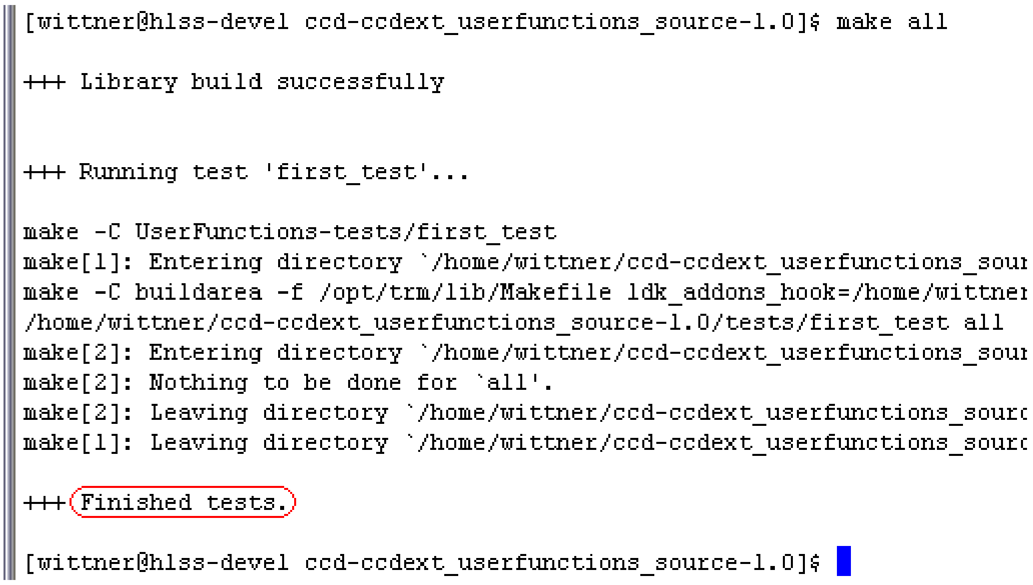
If the #define is missing, the compilation process of the test runner binary will fail with an error message stating that the user function is not implemented (Other error messages may follow, because some compilers unfortunately doesn't stop at the #error statement):



```
ccd2c -P0 -o out/testprocedure__64.c user/sourcecode/testprocedure__64.ccd
ccd2c :  0 ERROR(s), 0 WARNING(s)
cc -O3 -m32 -pipe -Wall -Wno-comment -Wno-unused-parameter -W -Wnested-externs -Wpointer-arith -Wcast-ali
out/testprocedure__64.c:94:3: error: #error "User function setDCPower() not implemented."
make: *** [user/bin/testprocedure__64.o] Error 1
```

3.2.3 Compiling the User Function Library

When you have finished implementing the user function, you may compile the user function library. This is done by invoking the **`make all`** command (using **`make`** only lists other available options):



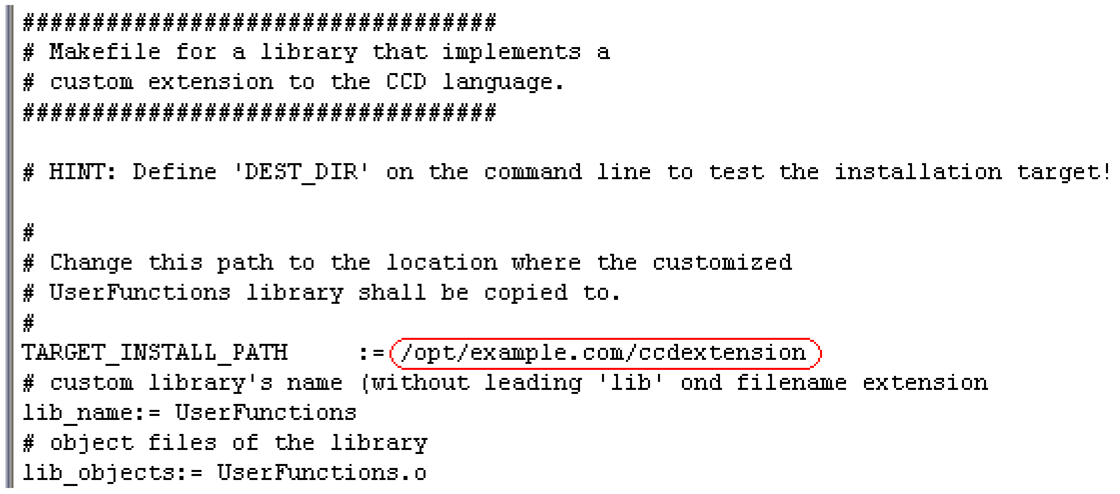
```
[wittner@hlss-devel ccd-ccdext_userfunctions_source-1.0]$ make all

+++ Library build successfully


+++ Running test 'first_test'...

make -C UserFunctions-tests/first_test
make[1]: Entering directory `/home/wittner/ccd-ccdext_userfunctions_sour
make -C buildarea -f /opt/trm/lib/Makefile ldk_addons_hook=/home/wittner
/home/wittner/ccd-ccdext_userfunctions_source-1.0/tests/first_test all
make[2]: Entering directory `/home/wittner/ccd-ccdext_userfunctions_sour
make[2]: Nothing to be done for `all'.
make[2]: Leaving directory `/home/wittner/ccd-ccdext_userfunctions_sourc
make[1]: Leaving directory `/home/wittner/ccd-ccdext_userfunctions_sourc

+++ Finished tests.

[wittner@hlss-devel ccd-ccdext_userfunctions_source-1.0]$
```

There will be much more messages (indicated by the dots), but the most important message is the final **+++ Finished tests** message. If this message does not appear, please check the messages starting from the bottom up to the top to find the cause of the problem.

The make command will also build an executable with the same compiler settings and libraries linked as when compiling and linking done by the test execution system. If make succeeds, you may install the library on the test execution system.

3.2.4 Installing the Newly Built Library

If everything compiles ok, you can install the newly built library on the test execution system. You need to adapt the **`Makefile`** in order to install the user function library into any suitable location within the file system. Change the path setting at the beginning of the Makefile like shown below:

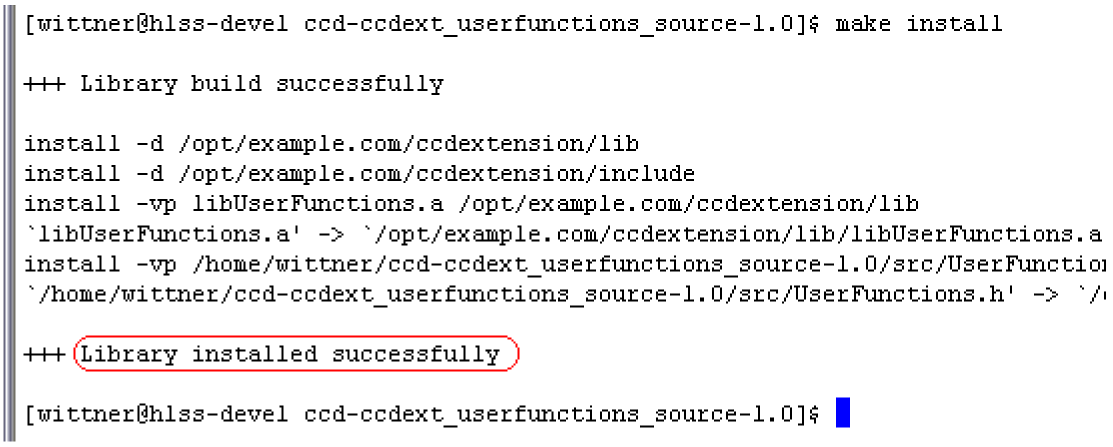


```
####################################
# Makefile for a library that implements a
# custom extension to the CCD language.
####################################

# HINT: Define 'DEST_DIR' on the command line to test the installation target!

#
# Change this path to the location where the customized
# UserFunctions library shall be copied to.
#
TARGET_INSTALL_PATH     :=/opt/example.com/ccdextension
# custom library's name (without leading 'lib' ond filename extension
lib_name:= UserFunctions
# object files of the library
lib_objects:= UserFunctions.o
```

Please note: The path settings shown above are only examples and should be customized.

Invoking the **`make install`** command will do the following actions:

- (re)build the user functions library
- copy the newly built library and header file into the specified target installation directory (TARGET_INSTALL_PATH).



```
[wittner@hlss-devel ccd-ccdext_userfunctions_source-1.0]$ make install

+++ Library build successfully

install -d /opt/example.com/ccdextension/lib
install -d /opt/example.com/ccdextension/include
install -vp libUserFunctions.a /opt/example.com/ccdextension/lib
`libUserFunctions.a' -> `/opt/example.com/ccdextension/lib/libUserFunctions.a
install -vp /home/wittner/ccd-ccdext_userfunctions_source-1.0/src/UserFunctio
`/home/wittner/ccd-ccdext_userfunctions_source-1.0/src/UserFunctions.h' -> `/

+++ Library installed successfully

[wittner@hlss-devel ccd-ccdext_userfunctions_source-1.0]$
```

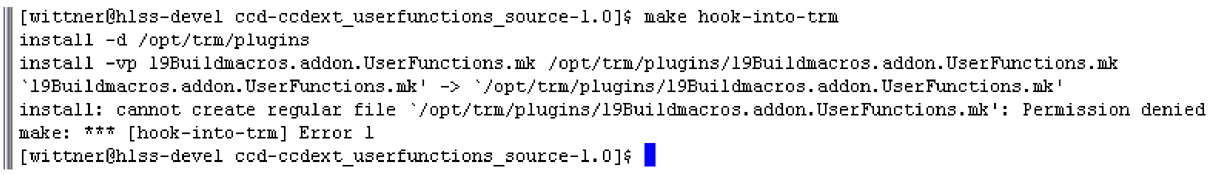
There may be more messages, but the most important message is the final **+++ Library installed successfully** message. If this message does not appear, please check the messages starting from the bottom up to the top to find the cause of the problem.

3.2.5 Activating the User Function Library

The user function library needs to be added into the compilation and linking process of the test execution system (e.g. the TRM system). A pre-configured plugin makefile for TRM will be generated and installed when running the **`make hook-into-trm`** command:

```
19Buildmacros.addon.userfunctions.mk
```

If you don't have root access rights, the installation process may be aborted when copying the generated plug-in makefile like shown below:



```
[wittner@hlss-devel ccd-ccdext_userfunctions_source-1.0]$ make hook-into-trm
install -d /opt/trm/plugins
install -vp 19Buildmacros.addon.UserFunctions.mk /opt/trm/plugins/19Buildmacros.addon.UserFunctions.mk
`19Buildmacros.addon.UserFunctions.mk' -> `/opt/trm/plugins/19Buildmacros.addon.UserFunctions.mk'
install: cannot create regular file `/opt/trm/plugins/19Buildmacros.addon.UserFunctions.mk': Permission denied
make: *** [hook-into-trm] Error 1
[wittner@hlss-devel ccd-ccdext_userfunctions_source-1.0]$
```

In this case, another user with root access rights may copy the readily configured plugin makefile manually into the following directory in order to add it to the TRM build process:

```
/opt/trm/plugins
```

4 CCDL Configuration Directory

The CCDL compiler requires some configuration files, which will be installed into the default location

```
/opt/razorcat/ccd/etc/config
```

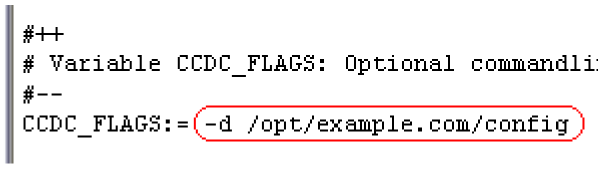
during the CCDL installation (the default files contain sample data). The following files are vital for proper operation of the CCDL compiler:

- ccd2c.conf (optional)
- constants.txt
- functions.txt
- macros.ccd
- modules.txt
- parameter.txt
- pins.txt
- units.txt

If you need to change these files (e.g. for an automatic synchronization with an SVN server) without having root access rights, you may specify an alternate directory for the CCDL configuration files and other options to the CCDL compiler within the following file:

```
/opt/trm/etc/ccdc-options.mk
```

You need to change the CCDC_FLAGS variable in order to specify options for the CCDL compiler (e.g. changing the path for the configuration files using the –d option):



```
#++
# Variable CCDC_FLAGS: Optional commandline arguments for the ccd compiler.
#--
CCDC_FLAGS:=-d /opt/example.com/config
```

5 Troubleshooting

5.1 #debug Statements

In case of any errors during the CCDL compilation process or when executing tests on the test execution system, you may (temporarily) add the following debug statements into the CCDL procedure (e.g. within the first line of the procedure text). These statements will activate more debug messages that may help to find the problem location.

- **`#debug compile`** enables debug messages concerning the compilation process of the CCDL compiler.
- **`#debug interface`** enables debug messages concerning the test system interface layer of the CCDL framework.
- **`#debug runtime`** enables debug messages concerning the runtime CCDL framework.

These messages will be available within the default log of the test execution system (e.g. within the **Log Messages** view of TOP).