

Common Test

Copyright © 2003-2023 Ericsson AB. All Rights Reserved.
Common Test 1.18.1
May 9, 2023

Copyright © 2003-2023 Ericsson AB. All Rights Reserved.
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. Ericsson AB. All Rights Reserved..

May 9, 2023

1 Common Test User's Guide

1.1 Introduction

1.1.1 Scope

Common Test is a portable application for automated testing. It is suitable for:

- Black-box testing of target systems of any type (that is, not necessarily implemented in Erlang). This is performed through standard O&M interfaces (such as SNMP, HTTP, CORBA, and Telnet) and, if necessary, through user-specific interfaces (often called test ports).
- White-box testing of Erlang/OTP programs. This is easily done by calling the target API functions directly from the test case functions.

Common Test also integrates use of the OTP *cover* tool in application Tools for code coverage analysis of Erlang/OTP programs.

Common Test executes test suite programs automatically, without operator interaction. Test progress and results are printed to logs in HTML format, easily browsed with a standard web browser. Common Test also sends notifications about progress and results through an OTP event manager to event handlers plugged in to the system. This way, users can integrate their own programs for, for example, logging, database storing, or supervision with Common Test.

Common Test provides libraries with useful support functions to fill various testing needs and requirements. There is, for example, support for flexible test declarations through test specifications. There is also support for central configuration and control of multiple independent test sessions (to different target systems) running in parallel.

1.1.2 Prerequisites

It is assumed that the reader is familiar with the Erlang programming language.

1.2 Common Test Basics

1.2.1 General

The Common Test framework is a tool that supports implementation and automated execution of test cases to any types of target systems. Common Test is the main tool being used in all testing- and verification activities that are part of Erlang/OTP system development and maintenance.

Test cases can be executed individually or in batches. Common Test also features a distributed testing mode with central control and logging. With this feature, multiple systems can be tested independently in one common session. This is useful, for example, when running automated large-scale regression tests.

The System Under Test (SUT) can consist of one or more target nodes. Common Test contains a generic test server that, together with other test utilities, is used to perform test case execution. The tests can be started from a GUI, from the OS shell, or from an Erlang shell. **Test suites** are files (Erlang modules) that contain the **test cases** (Erlang functions) to be executed. **Support modules** provide functions that the test cases use to do the tests.

In a black-box testing scenario, Common Test-based test programs connect to the target system(s) through standard O&M and CLI protocols. Common Test provides implementations of, and wrapper interfaces to, some of these protocols (most of which exist as standalone components and applications in OTP). The wrappers simplify configuration and add verbosity for logging purposes. Common Test is continously extended with useful support

modules. However, notice that it is a straightforward task to use any Erlang/OTP component for testing purposes with Common Test, without needing a Common Test wrapper for it. It is as simple as calling Erlang functions. A number of target-independent interfaces are supported in Common Test, such as Generic Telnet and FTP. These can be specialized or used directly for controlling instruments, traffic load generators, and so on.

Common Test is also a very useful tool for white-box testing Erlang code (for example, module testing), as the test programs can call exported Erlang functions directly. There is very little overhead required for implementing basic test suites and executing simple tests. For black-box testing Erlang software, Erlang RPC and standard O&M interfaces can be used for example.

A test case can handle several connections to one or more target systems, instruments, and traffic generators in parallel to perform the necessary actions for a test. The handling of many connections in parallel is one of the major strengths of Common Test, thanks to the efficient support for concurrency in the Erlang runtime system, which Common Test users can take great advantage of.

1.2.2 Test Suite Organisation

Test suites are organized in test directories and each test suite can have a separate data directory. Typically, these files and directories are version-controlled similar to other forms of source code (possibly by a version control system like GIT or Subversion). However, Common Test does not itself put any requirements on (or has any awareness of) possible file and directory versions.

1.2.3 Support Libraries

Support libraries contain functions that are useful for all test suites, or for test suites in a specific functional area or subsystem. In addition to the general support libraries provided by the Common Test framework, and the various libraries and applications provided by Erlang/OTP, there can also be a need for customized (user specific) support libraries.

1.2.4 Suites and Test Cases

Testing is performed by running test suites (sets of test cases) or individual test cases. A test suite is implemented as an Erlang module named <suite_name>_SUITE.erl which contains a number of test cases. A test case is an Erlang function that tests one or more things. The test case is the smallest unit that the Common Test test server deals with.

Sets of test cases, called test case groups, can also be defined. A test case group can have execution properties associated with it. Execution properties specify if the test cases in the group are to be executed in random order, in parallel, or in sequence, and if the execution of the group is to be repeated. Test case groups can also be nested (that is, a group can, besides test cases, contain subgroups).

Besides test cases and groups, the test suite can also contain configuration functions. These functions are meant to be used for setting up (and verifying) environment and state in the SUT (and/or the Common Test host node), required for the tests to execute correctly. Examples of operations are: Opening a connection to the SUT, initializing a database, running an installation script, and so on. Configuration can be performed per suite, per test case group, and per individual test case.

The test suite module must conform to a *callback interface* specified by the Common Test test server. For details, see section *Writing Test Suites*.

A test case is considered successful if it returns to the caller, no matter what the returned value is. However, a few return values have special meaning as follows:

- {skip,Reason} indicates that the test case is skipped.
- {comment,Comment} prints a comment in the log for the test case.
- {save_config,Config} makes the Common Test test server pass Config to the next test case.

A test case failure is specified as a runtime error (a crash), no matter what the reason for termination is. If you use Erlang pattern matching effectively, you can take advantage of this property. The result is concise and readable test case functions that look much more like scripts than actual programs. A simple example:

```
session(_Config) ->
    {started,ServerId} = my_server:start(),
    {clients,[]} = my_server:get_clients(ServerId),
    MyId = self(),
    connected = my_server:connect(ServerId, MyId),
    {clients,[MyId]} = my_server:get_clients(ServerId),
    disconnected = my_server:disconnect(ServerId, MyId),
    {clients,[]} = my_server:get_clients(ServerId),
    stopped = my_server:stop(ServerId).
```

As a test suite runs, all information (including output to `stdout`) is recorded in many different log files. A minimum of information is displayed in the user console (only start and stop information, plus a note for each failed test case).

The result from each test case is recorded in a dedicated HTML log file, created for the particular test run. An overview page displays each test case represented by a table row showing total execution time, if the case was successful, failed, or skipped, plus an optional user comment. For a failed test case, the reason for termination is also printed in the comment field. The overview page has a link to each test case log file, providing simple navigation with any standard HTML browser.

1.2.5 External Interfaces

The `Common Test` test server requires that the test suite defines and exports the following mandatory or optional callback functions:

`all()`

Returns a list of all test cases and groups in the suite. (Mandatory)

`suite()`

Information function used to return properties for the suite. (Optional)

`groups()`

For declaring test case groups. (Optional)

`init_per_suite(Config)`

Suite level configuration function, executed before the first test case. (Optional)

`end_per_suite(Config)`

Suite level configuration function, executed after the last test case. (Optional)

`group(GroupName)`

Information function used to return properties for a test case group. (Optional)

`init_per_group(GroupName, Config)`

Configuration function for a group, executed before the first test case. (Optional)

`end_per_group(GroupName, Config)`

Configuration function for a group, executed after the last test case. (Optional)

`init_per_testcase(TestCase, Config)`

Configuration function for a testcase, executed before each test case. (Optional)

`end_per_testcase(TestCase, Config)`

Configuration function for a testcase, executed after each test case. (Optional)

For each test case, the `Common Test` test server expects the following functions:

Testcasename()

Information function that returns a list of test case properties. (Optional)

Testcasename(Config)

The test case function.

1.3 Getting Started

1.3.1 Introduction for Newcomers

The purpose of this section is to let the newcomer get started in quickly writing and executing some first simple tests with a "learning by example" approach. Most explanations are left for later sections. If you are not much into "learning by example" and prefer more technical details, go ahead and skip to the next section.

This section demonstrates how simple it is to write a basic (yet for many module testing purposes, often sufficiently complex) test suite and execute its test cases. This is not necessarily obvious when you read the remaining sections in this User's Guide.

Note:

To understand what is discussed and examplified here, we recommended you to first read section *Common Test Basics*.

1.3.2 Test Case Execution

Execution of test cases is handled as follows:

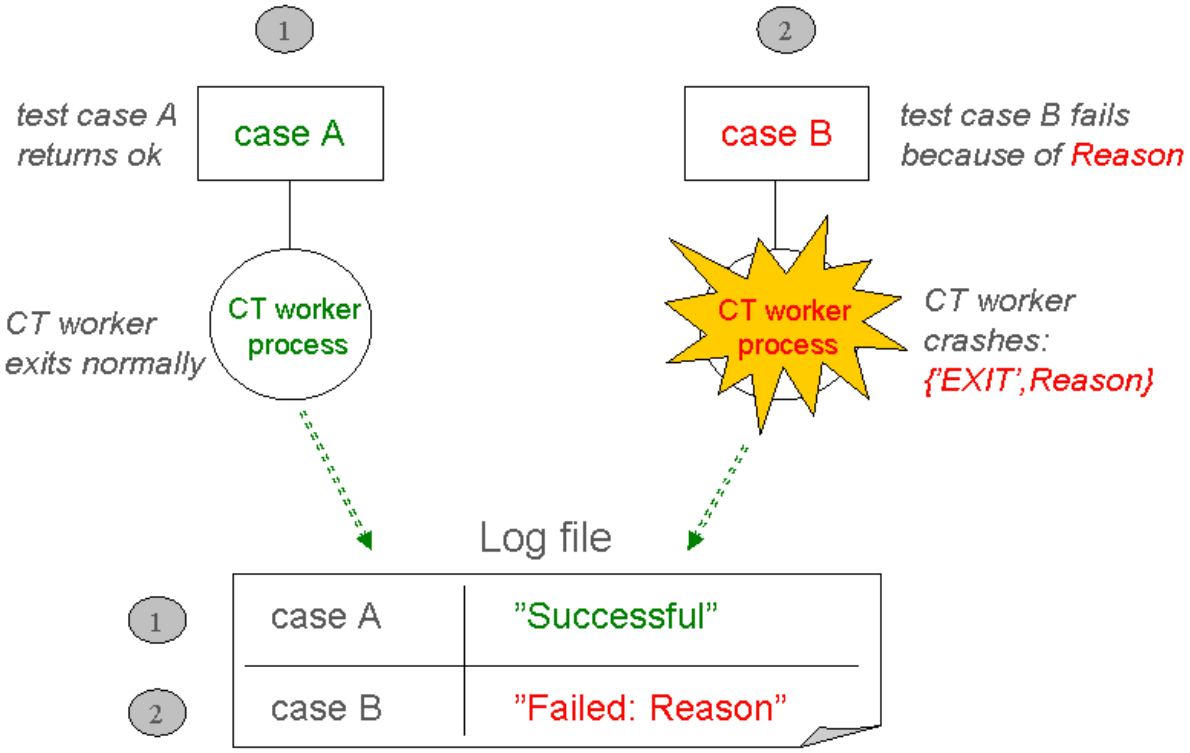


Figure 3.1: Successful and Unsuccessful Test Case Execution

For each test case that `Common Test` is ordered to execute, it spawns a dedicated process on which the test case function starts running. (In parallel to the test case process, an idle waiting timer process is started, which is linked to the test case process. If the timer process runs out of waiting time, it sends an exit signal to terminate the test case process. This is called a **timetrap**).

In scenario 1, the test case process terminates normally after `case A` has finished executing its test code without detecting any errors. The test case function returns a value and `Common Test` logs the test case as successful.

In scenario 2, an error is detected during test `case B` execution. This causes the test `case B` function to generate an exception and, as a result, the test case process exits with reason other than normal. `Common Test` logs this as an unsuccessful (Failed) test case.

As you can understand from the illustration, `Common Test` requires a test case to generate a runtime error to indicate failure (for example, by causing a bad match error or by calling `exit/1`, preferably through the help function *`ct:fail/1,2`*). A successful execution is indicated by a normal return from the test case function.

1.3.3 A Simple Test Suite

As shown in section *Common Test Basics*, the test suite module implements *callback functions* (mandatory or optional) for various purposes, for example:

- Init/end configuration function for the test suite
- Init/end configuration function for a test case
- Init/end configuration function for a test case group
- Test cases

The configuration functions are optional. The following example is a test suite without configuration functions, including one simple test case, to check that module `mymod` exists (that is, can be successfully loaded by the code server):

```
-module(my1st_SUITE).
-compile(export_all).

all() ->
    [mod_exists].

mod_exists(_) ->
    {module,mymod} = code:load_file(mymod).
```

If the operation fails, a bad match error occurs that terminates the test case.

1.3.4 A Test Suite with Configuration Functions

If you need to perform configuration operations to run your test, you can implement configuration functions in your suite. The result from a configuration function is configuration data, or `Config`. This is a list of key-value tuples that get passed from the configuration function to the test cases (possibly through configuration functions on "lower level"). The data flow looks as follows:

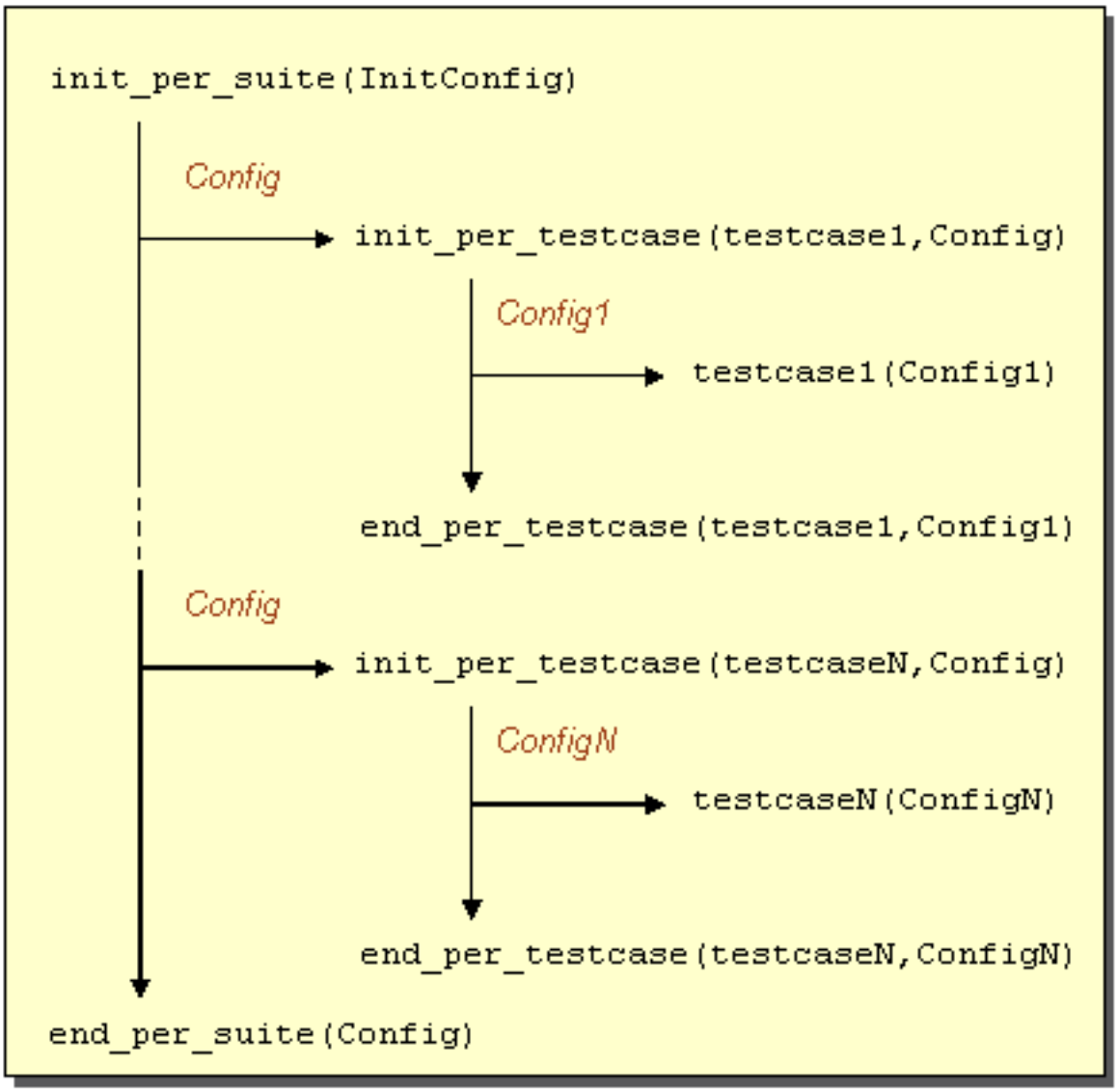


Figure 3.2: Configuration Data Flow in a Suite

The following example shows a test suite that uses configuration functions to open and close a log file for the test cases (an operation that is unnecessary and irrelevant to perform by each test case):

```
-module(check_log_SUITE).
-export([all/0, init_per_suite/1, end_per_suite/1]).
-export([check_restart_result/1, check_no_errors/1]).

-define(value(Key,Config), proplists:get_value(Key,Config)).

all() -> [check_restart_result, check_no_errors].

init_per_suite(InitConfigData) ->
    [{logref,open_log()} | InitConfigData].

end_per_suite(ConfigData) ->
    close_log(?value(logref, ConfigData)).

check_restart_result(ConfigData) ->
    TestData = read_log(restart, ?value(logref, ConfigData)),
    {match,_Line} = search_for("restart successful", TestData).

check_no_errors(ConfigData) ->
    TestData = read_log(all, ?value(logref, ConfigData)),
    case search_for("error", TestData) of
 {match,Line} -> ct:fail({error_found_in_log,Line});
 nomatch -> ok
    end.
```

The test cases verify, by parsing a log file, that our SUT has performed a successful restart and that no unexpected errors are printed.

To execute the test cases in the recent test suite, type the following on the UNIX/Linux command line (assuming that the suite module is in the current working directory):

```
$ ct_run -dir .
```

or:

```
$ ct_run -suite check_log_SUITE
```

To use the Erlang shell to run our test, you can evaluate the following call:

```
1> ct:run_test([{dir, "."}]).
```

or:

```
1> ct:run_test([{suite, "check_log_SUITE"}]).
```

The result from running the test is printed in log files in HTML format (stored in unique log directories on a different level). The following illustration shows the log file structure:

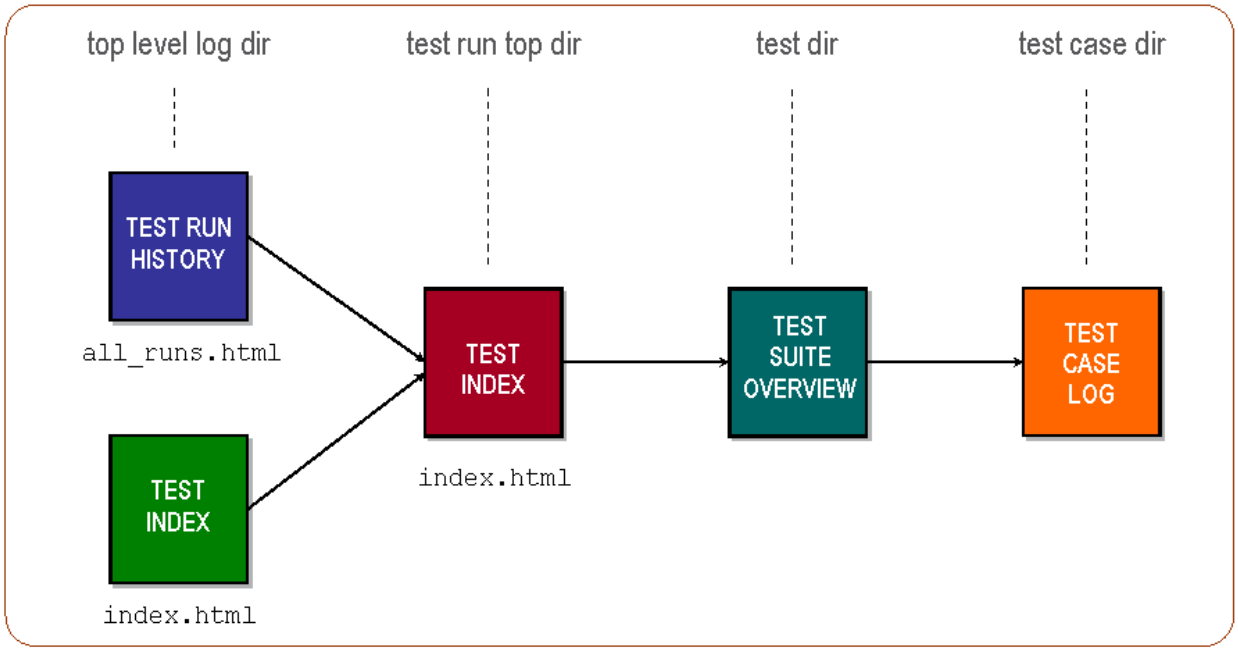


Figure 3.3: HTML Log File Structure

1.3.5 Questions and Answers

Here follows some questions that you might have after reading this section with corresponding tips and links to the answers:

- **Question:** "How and where can I specify variable data for my tests that must not be hard-coded in the test suites (such as hostnames, addresses, and user login data)?"

  Answer: See section *External Configuration Data*.
- **Question:** "Is there a way to declare different tests and run them in one session without having to write my own scripts? Also, can such declarations be used for regression testing?"

  Answer: See section *Test Specifications* in section Running Tests and Analyzing Results.
- **Question:** "Can test cases and/or test runs be automatically repeated?"

  Answer: Learn more about *Test Case Groups* and read about start flags/options in section *Running Tests* and in the Reference Manual.
- **Question:** "Does `Common Test` execute my test cases in sequence or in parallel?"

  Answer: See *Test Case Groups* in section Writing Test Suites.
- **Question:** "What is the syntax for timetraps (mentioned earlier), and how do I set them?"

  Answer: This is explained in the *Timetrap Time-Outs* part of section Writing Test Suites.
- **Question:** "What functions are available for logging and printing?"

  Answer: See *Logging* in section Writing Test Suites.
- **Question:** "I need data files for my tests. Where do I store them preferably?"

  Answer: See *Data and Private Directories*.
- **Question:** "Can I start with a test suite example, please?"

  Answer: *Welcome!*

You probably want to get started on your own first test suites now, while at the same time digging deeper into the Common Test User's Guide and Reference Manual. There are much more to learn about the things that have been introduced in this section. There are also many other useful features to learn, so please continue to the other sections and have fun.

1.4 Installation

1.4.1 General Information

The two main interfaces for running tests with Common Test are an executable program named *ct_run* and the Erlang module *ct*. ct_run is compiled for the underlying operating system (for example, Unix/Linux or Windows) during the build of the Erlang/OTP system, and is installed automatically with other executable programs in the top level bin directory of Erlang/OTP. The ct interface functions can be called from the Erlang shell, or from any Erlang function, on any supported platform.

The Common Test application is installed with the Erlang/OTP system. No extra installation step is required to start using Common Test through the ct_run executable program, and/or the interface functions in the ct module.

1.5 Writing Test Suites

1.5.1 Support for Test Suite Authors

The *ct* module provides the main interface for writing test cases. This includes for example, the following:

- Functions for printing and logging
- Functions for reading configuration data
- Function for terminating a test case with error reason
- Function for adding comments to the HTML overview page

For details about these functions, see module *ct*.

The Common Test application also includes other modules named ct_<component>, which provide various support, mainly simplified use of communication protocols such as RPC, SNMP, FTP, Telnet, and others.

1.5.2 Test Suites

A test suite is an ordinary Erlang module that contains test cases. It is recommended that the module has a name on the form *_SUITE.erl. Otherwise, the directory and auto compilation function in Common Test cannot locate it (at least not by default).

It is also recommended that the ct.hrl header file is included in all test suite modules.

Each test suite module must export function *all/0*, which returns the list of all test case groups and test cases to be executed in that module.

The callback functions to be implemented by the test suite are all listed in module *common_test* . They are also described in more detail later in this User's Guide.

1.5.3 Init and End per Suite

Each test suite module can contain the optional configuration functions *init_per_suite/1* and *end_per_suite/1*. If the init function is defined, so must the end function be.

If init_per_suite exists, it is called initially before the test cases are executed. It typically contains initializations common for all test cases in the suite, which are only to be performed once. init_per_suite is recommended for

setting up and verifying state and environment on the System Under Test (SUT) or the `Common Test` host node, or both, so that the test cases in the suite executes correctly. The following are examples of initial configuration operations:

- Opening a connection to the SUT
- Initializing a database
- Running an installation script

`end_per_suite` is called as the final stage of the test suite execution (after the last test case has finished). The function is meant to be used for cleaning up after `init_per_suite`.

`init_per_suite` and `end_per_suite` execute on dedicated Erlang processes, just like the test cases do. The result of these functions is however not included in the test run statistics of successful, failed, and skipped cases.

The argument to `init_per_suite` is `Config`, that is, the same key-value list of runtime configuration data that each test case takes as input argument. `init_per_suite` can modify this parameter with information that the test cases need. The possibly modified `Config` list is the return value of the function.

If `init_per_suite` fails, all test cases in the test suite are skipped automatically (so called **auto skipped**), including `end_per_suite`.

Notice that if `init_per_suite` and `end_per_suite` do not exist in the suite, `Common Test` calls dummy functions (with the same names) instead, so that output generated by hook functions can be saved to the log files for these dummies. For details, see *Common Test Hooks*.

1.5.4 Init and End per Test Case

Each test suite module can contain the optional configuration functions *`init_per_testcase/2`* and *`end_per_testcase/2`*. If the init function is defined, so must the end function be.

If `init_per_testcase` exists, it is called before each test case in the suite. It typically contains initialization that must be done for each test case (analog to `init_per_suite` for the suite).

`end_per_testcase/2` is called after each test case has finished, enabling cleanup after `init_per_testcase`.

The first argument to these functions is the name of the test case. This value can be used with pattern matching in function clauses or conditional expressions to choose different initialization and cleanup routines for different test cases, or perform the same routine for many, or all, test cases.

The second argument is the `Config` key-value list of runtime configuration data, which has the same value as the list returned by `init_per_suite`. `init_per_testcase/2` can modify this parameter or return it "as is". The return value of `init_per_testcase/2` is passed as parameter `Config` to the test case itself.

The return value of `end_per_testcase/2` is ignored by the test server, with exception of the *`save_config`* and `fail` tuple.

`end_per_testcase` can check if the test case was successful. (which in turn can determine how cleanup is to be performed). This is done by reading the value tagged with `tc_status` from `Config`. The value is one of the following:

- `ok`
- `{failed,Reason}`

  where `Reason` is `timetrap_timeout`, information from `exit/1`, or details of a runtime error
- `{skipped,Reason}`

  where `Reason` is a user-specific term

Function `end_per_testcase/2` is even called if a test case terminates because of a call to *`ct:abort_current_testcase/1`*, or after a timetrap time-out. However, `end_per_testcase` then executes on a different process than the test case function. In this situation, `end_per_testcase` cannot change the reason for test case termination by returning `{fail,Reason}` or save data with `{save_config,Data}`.

The test case is skipped in the following two cases:

- If `init_per_testcase` crashes (called **auto skipped**).
- If `init_per_testcase` returns a tuple `{skip,Reason}` (called **user skipped**).

The test case can also be marked as failed without executing it by returning a tuple `{fail,Reason}` from `init_per_testcase`.

Note:

If `init_per_testcase` crashes, or returns `{skip,Reason}` or `{fail,Reason}`, function `end_per_testcase` is not called.

If it is determined during execution of `end_per_testcase` that the status of a successful test case is to be changed to failed, `end_per_testcase` can return the tuple `{fail,Reason}` (where `Reason` describes why the test case fails).

As `init_per_testcase` and `end_per_testcase` execute on the same Erlang process as the test case, printouts from these configuration functions are included in the test case log file.

1.5.5 Test Cases

The smallest unit that the test server is concerned with is a test case. Each test case can test many things, for example, make several calls to the same interface function with different parameters.

The author can choose to put many or few tests into each test case. Some things to keep in mind follows:

- Many small test cases tend to result in extra, and possibly duplicated code, as well as slow test execution because of large overhead for initializations and cleanups. Avoid duplicated code, for example, by using common help functions. Otherwise, the resulting suite becomes difficult to read and understand, and expensive to maintain.
- Larger test cases make it harder to tell what went wrong if it fails. Also, large portions of test code risk being skipped when errors occur.
- Readability and maintainability suffer when test cases become too large and extensive. It is not certain that the resulting log files reflect very well the number of tests performed.

The test case function takes one argument, `Config`, which contains configuration information such as `data_dir` and `priv_dir`. (For details about these, see section *Data and Private Directories*. The value of `Config` at the time of the call, is the same as the return value from `init_per_testcase`, mentioned earlier.

Note:

The test case function argument `Config` is not to be confused with the information that can be retrieved from the configuration files (using *`ct:get_config/1/2`*). The test case argument `Config` is to be used for runtime configuration of the test suite and the test cases, while configuration files are to contain data related to the SUT. These two types of configuration data are handled differently.

As parameter `Config` is a list of key-value tuples, that is, a data type called a property list, it can be handled by the *`proplists`* module. A value can, for example, be searched for and returned with function *`proplists:get_value/2`*. Also, or alternatively, the general *`lists`* module contains useful functions. Normally, the only operations performed on `Config` is insert (adding a tuple to the head of the list) and lookup. `Common Test` provides a simple macro named `?config`, which returns a value of an item in `Config` given the key (exactly like `proplists:get_value`). Example: `PrivDir = ?config(priv_dir, Config)`.

If the test case function crashes or exits purposely, it is considered **failed**. If it returns a value (no matter what value), it is considered successful. An exception to this rule is the return value `{skip,Reason}`. If this tuple is returned, the test case is considered skipped and is logged as such.

If the test case returns the tuple `{comment,Comment}`, the case is considered successful and `Comment` is printed in the overview log file. This is equal to calling *`ct:comment(Comment)`*.

1.5.6 Test Case Information Function

For each test case function there can be an extra function with the same name but without arguments. This is the test case information function. It is expected to return a list of tagged tuples that specifies various properties regarding the test case.

The following tags have special meaning:

`timetrap`

Sets the maximum time the test case is allowed to execute. If this time is exceeded, the test case fails with reason `timetrap_timeout`. Notice that `init_per_testcase` and `end_per_testcase` are included in the timetrap time. For details, see section *Timetrap Time-Outs*.

`userdata`

Specifies any data related to the test case. This data can be retrieved at any time using the *`ct:userdata/3`* utility function.

`silent_connections`

For details, see section *Silent Connections*.

`require`

Specifies configuration variables required by the test case. If the required configuration variables are not found in any of the test system configuration files, the test case is skipped.

A required variable can also be given a default value to be used if the variable is not found in any configuration file. To specify a default value, add a tuple on the form `{default_config,ConfigVariableName,Value}` to the test case information list (the position in the list is irrelevant).

Examples:

```
testcase1() ->
    [{require, ftp},
     {default_config, ftp, [{ftp, "my_ftp_host"},
                            {username, "aladdin"},
                            {password, "sesame"}]}}].
```

```
testcase2() ->
    [{require, unix_telnet, unix},
     {require, {unix, [telnet, username, password]}},
     {default_config, unix, [{telnet, "my_telnet_host"},
                             {username, "aladdin"},
                             {password, "sesame"}]}}].
```

For more information about `require`, see section *Requiring and Reading Configuration Data* in section External Configuration Data and function *`ct:require/1/2`*.

Note:

Specifying a default value for a required variable can result in a test case always getting executed. This might not be a desired behavior.

If `timetrap` or `require`, or both, is not set specifically for a particular test case, default values specified by function *`suite/0`* are used.

Tags other than the earlier mentioned are ignored by the test server.

An example of a test case information function follows:

```
reboot_node() ->
    [
     {timetrap,{seconds,60}},
     {require,interfaces},
     {userdata,
         [{description,"System Upgrade: RpuAddition Normal RebootNode"},
          {fts,"http://someserver.ericsson.se/test_doc4711.pdf"}]}
    ].
```

1.5.7 Test Suite Information Function

Function *`suite/0`* can, for example, be used in a test suite module to set a default `timetrap` value and to `require` external configuration data. If a test case, or a group information function also specifies any of the information tags, it overrides the default values set by `suite/0`. For details, see *Test Case Information Function* and *Test Case Groups*.

The following options can also be specified with the suite information list:

- `stylesheet`, see *HTML Style Sheets*
- `userdata`, see *Test Case Information Function*
- `silent_connections`, see *Silent Connections*

An example of the suite information function follows:

```
suite() ->
    [
     {timetrap,{minutes,10}},
     {require,global_names},
     {userdata,[{info,"This suite tests database transactions."}]},
     {silent_connections,[telnet]},
     {stylesheet,"db_testing.css"}
    ].
```

1.5.8 Test Case Groups

A test case group is a set of test cases sharing configuration functions and execution properties. Test case groups are defined by function *`groups/0`* according to the following syntax:

```
groups() -> GroupDefs

Types:

GroupDefs = [GroupDef]
GroupDef = {GroupName,Properties,GroupsAndTestCases}
GroupName = atom()
GroupsAndTestCases = [GroupDef | {group,GroupName} | TestCase |
                      {testcase,TestCase,TCRepeatProps}]
TestCase = atom()
TCRepeatProps = [{repeat,N} | {repeat_until_ok,N} | {repeat_until_fail,N}]
```

`GroupName` is the name of the group and must be unique within the test suite module. Groups can be nested, by including a group definition within the `GroupsAndTestCases` list of another group. `Properties` is the list of execution properties for the group. The possible values are as follows:

```
Properties = [parallel | sequence | Shuffle | {GroupRepeatType,N}]
Shuffle = shuffle | {shuffle,Seed}
Seed = {integer(),integer(),integer()}
GroupRepeatType = repeat | repeat_until_all_ok | repeat_until_all_fail |
                  repeat_until_any_ok | repeat_until_any_fail
N = integer() | forever
```

Explanations:

`parallel`

`Common Test` executes all test cases in the group in parallel.

`sequence`

The cases are executed in a sequence as described in section *Sequences* in section Dependencies Between Test Cases and Suites.

`shuffle`

The cases in the group are executed in random order.

`repeat, repeat_until_*`

Orders `Common Test` to repeat execution of all the cases in the group a given number of times, or until any, or all, cases fail or succeed.

Example:

```
groups() -> [{group1, [parallel], [test1a,test1b]},
             {group2, [shuffle,sequence], [test2a,test2b,test2c]}].
```

To specify in which order groups are to be executed (also with respect to test cases that are not part of any group), add tuples on the form `{group,GroupName}` to the `all/0` list.

Example:

```
all() -> [testcase1, {group,group1}, {testcase,testcase2,[{repeat,10}]}, {group,group2}].
```

Execution properties with a group tuple in `all/0`: `{group,GroupName,Properties}` can also be specified. These properties override those specified in the group definition (see `groups/0` earlier). This way, the same set of tests can be run, but with different properties, without having to make copies of the group definition in question.

If a group contains subgroups, the execution properties for these can also be specified in the group tuple: `{group,GroupName,Properties,SubGroups}` Where, `SubGroups` is a list of tuples, `{GroupName,Properties}` or `{GroupName,Properties,SubGroups}` representing the subgroups. Any subgroups defined in `group/0` for a group, that are not specified in the `SubGroups` list, executes with their predefined properties.

Example:

```
groups() -> {tests1, [], [{tests2, [], [t2a,t2b]},
                          {tests3, [], [t31,t3b]}]}.
```

To execute group `tests1` twice with different properties for `tests2` each time:

```
all() ->
   [{group, tests1, default, [{tests2, [parallel]}]},
    {group, tests1, default, [{tests2, [shuffle,{repeat,10}]}]}].
```

This is equivalent to the following specification:

```
all() ->
   [{group, tests1, default, [{tests2, [parallel]},
                              {tests3, default}]},
    {group, tests1, default, [{tests2, [shuffle,{repeat,10}]},
                              {tests3, default}]}].
```

Value `default` states that the predefined properties are to be used.

The following example shows how to override properties in a scenario with deeply nested groups:

```
groups() ->
   [{tests1, [], [{group, tests2}]},
    {tests2, [], [{group, tests3}]},
    {tests3, [{repeat,2}], [t3a,t3b,t3c]}].

all() ->
   [{group, tests1, default,
     [{tests2, default,
       [{tests3, [parallel,{repeat,100}]}]}]}].
```

The described syntax can also be used in test specifications to change group properties at the time of execution, without having to edit the test suite. For more information, see section *Test Specifications* in section Running Tests and Analyzing Results.

As illustrated, properties can be combined. If, for example, `shuffle`, `repeat_until_any_fail`, and `sequence` are all specified, the test cases in the group are executed repeatedly, and in random order, until a test case fails. Then execution is immediately stopped and the remaining cases are skipped.

Before execution of a group begins, the configuration function *`init_per_group(GroupName, Config)`* is called. The list of tuples returned from this function is passed to the test cases in the usual manner by argument `Config`. `init_per_group/2` is meant to be used for initializations common for the test cases in the group. After execution of the group is finished, function *`end_per_group(GroupName, Config)`* is called. This function is meant to be used for cleaning up after `init_per_group/2`. If the init function is defined, so must the end function be.

Whenever a group is executed, if `init_per_group` and `end_per_group` do not exist in the suite, `Common Test` calls dummy functions (with the same names) instead. Output generated by hook functions are saved to the log files for these dummies. For more information, see section *Manipulating Tests* in section Common Test Hooks.

Note:

`init_per_testcase/2` and `end_per_testcase/2` are always called for each individual test case, no matter if the case belongs to a group or not.

The properties for a group are always printed in the top of the HTML log for `init_per_group/2`. The total execution time for a group is included at the bottom of the log for `end_per_group/2`.

Test case groups can be nested so sets of groups can be configured with the same `init_per_group/2` and `end_per_group/2` functions. Nested groups can be defined by including a group definition, or a group name reference, in the test case list of another group.

Example:

```
groups() -> [{group1, [shuffle], [test1a,
                                  {group2, [], [test2a,test2b]},
                                  test1b]},
             {group3, [], [{group,group4},
                           {group,group5}]},
             {group4, [parallel], [test4a,test4b]},
             {group5, [sequence], [test5a,test5b,test5c]}].
```

In the previous example, if `all/0` returns group name references in the order `[{group,group1}, {group,group3}]`, the order of the configuration functions and test cases becomes the following (notice that `init_per_testcase/2` and `end_per_testcase/2:` are also always called, but not included in this example for simplification):

```
init_per_group(group1, Config) -> Config1  (*)
     test1a(Config1)
     init_per_group(group2, Config1) -> Config2
          test2a(Config2), test2b(Config2)
     end_per_group(group2, Config2)
     test1b(Config1)
end_per_group(group1, Config1)
init_per_group(group3, Config) -> Config3
     init_per_group(group4, Config3) -> Config4
          test4a(Config4), test4b(Config4)  (**)
     end_per_group(group4, Config4)
     init_per_group(group5, Config3) -> Config5
          test5a(Config5), test5b(Config5), test5c(Config5)
     end_per_group(group5, Config5)
end_per_group(group3, Config3)
```

(*) The order of test case `test1a`, `test1b`, and `group2` is undefined, as `group1` has a shuffle property.

(**) These cases are not executed in order, but in parallel.

Properties are not inherited from top-level groups to nested subgroups. For instance, in the previous example, the test cases in `group2` are not executed in random order (which is the property of `group1`).

1.5.9 Parallel Property and Nested Groups

If a group has a parallel property, its test cases are spawned simultaneously and get executed in parallel. However, a test case is not allowed to execute in parallel with `end_per_group/2`, which means that the time to execute a parallel group is equal to the execution time of the slowest test case in the group. A negative side effect of running test cases in parallel is that the HTML summary pages are not updated with links to the individual test case logs until function `end_per_group/2` for the group has finished.

A group nested under a parallel group starts executing in parallel with previous (parallel) test cases (no matter what properties the nested group has). However, as test cases are never executed in parallel with `init_per_group/2` or `end_per_group/2` of the same group, it is only after a nested group has finished that remaining parallel cases in the previous group become spawned.

1.5.10 Parallel Test Cases and I/O

A parallel test case has a private I/O server as its group leader. (For a description of the group leader concept, see *ERTS*). The central I/O server process, which handles the output from regular test cases and configuration functions, does not respond to I/O messages during execution of parallel groups. This is important to understand to avoid certain traps, like the following:

If a process, `P`, is spawned during execution of, for example, `init_per_suite/1`, it inherits the group leader of the `init_per_suite` process. This group leader is the central I/O server process mentioned earlier. If, at a later

time, **during parallel test case execution**, some event triggers process P to call io:format/1/2, that call never returns (as the group leader is in a non-responsive state) and causes P to hang.

1.5.11 Repeated Groups

A test case group can be repeated a certain number of times (specified by an integer) or indefinitely (specified by forever). The repetition can also be stopped too early if any or all cases fail or succeed, that is, if any of the properties repeat_until_any_fail, repeat_until_any_ok, repeat_until_all_fail, or repeat_until_all_ok is used. If the basic repeat property is used, status of test cases is irrelevant for the repeat operation.

The status of a subgroup can be returned (ok or failed), to affect the execution of the group on the level above. This is accomplished by, in end_per_group/2, looking up the value of tc_group_properties in the Config list and checking the result of the test cases in the group. If status failed is to be returned from the group as a result, end_per_group/2 is to return the value {return_group_result,failed}. The status of a subgroup is taken into account by Common Test when evaluating if execution of a group is to be repeated or not (unless the basic repeat property is used).

The value of tc_group_properties is a list of status tuples, each with the key ok, skipped, and failed. The value of a status tuple is a list with names of test cases that have been executed with the corresponding status as result.

The following is an example of how to return the status from a group:

```
end_per_group(_Group, Config) ->
    Status = ?config(tc_group_result, Config),
    case proplists:get_value(failed, Status) of
        [] ->                                   % no failed cases
            {return_group_result,ok};
        _Failed ->                              % one or more failed
            {return_group_result,failed}
    end.
```

It is also possible, in end_per_group/2, to check the status of a subgroup (maybe to determine what status the current group is to return). This is as simple as illustrated in the previous example, only the group name is stored in a tuple {group_result,GroupName}, which can be searched for in the status lists.

Example:

```
end_per_group(group1, Config) ->
    Status = ?config(tc_group_result, Config),
    Failed = proplists:get_value(failed, Status),
    case lists:member({group_result,group2}, Failed) of
          true ->
              {return_group_result,failed};
          false ->
              {return_group_result,ok}
    end;
...
```

Note:

When a test case group is repeated, the configuration functions init_per_group/2 and end_per_group/2 are also always called with each repetition.

1.5.12 Shuffled Test Case Order

The order in which test cases in a group are executed is under normal circumstances the same as the order specified in the test case list in the group definition. With property `shuffle` set, however, `Common Test` instead executes the test cases in random order.

You can provide a seed value (a tuple of three integers) with the shuffle property `{shuffle,Seed}`. This way, the same shuffling order can be created every time the group is executed. If no seed value is specified, `Common Test` creates a "random" seed for the shuffling operation (using the return value of `erlang:timestamp/0`). The seed value is always printed to the `init_per_group/2` log file so that it can be used to recreate the same execution order in a subsequent test run.

Note:

If a shuffled test case group is repeated, the seed is not reset between turns.

If a subgroup is specified in a group with a `shuffle` property, the execution order of this subgroup in relation to the test cases (and other subgroups) in the group, is random. The order of the test cases in the subgroup is however not random (unless the subgroup has a `shuffle` property).

1.5.13 Group Information Function

The test case group information function, `group(GroupName)`, serves the same purpose as the suite- and test case information functions previously described. However, the scope for the group information function, is all test cases and subgroups in the group in question (`GroupName`).

Example:

```
group(connection_tests) ->
   [{require,login_data},
    {timetrap,1000}].
```

The group information properties override those set with the suite information function, and can in turn be overridden by test case information properties. For a list of valid information properties and more general information, see the *Test Case Information Function*.

1.5.14 Information Functions for Init- and End-Configuration

Information functions can also be used for functions `init_per_suite`, `end_per_suite`, `init_per_group`, and `end_per_group`, and they work the same way as with the *Test Case Information Function*. This is useful, for example, for setting timetraps and requiring external configuration data relevant only for the configuration function in question (without affecting properties set for groups and test cases in the suite).

The information function `init/end_per_suite()` is called for `init/end_per_suite(Config)`, and information function `init/end_per_group(GroupName)` is called for `init/end_per_group(GroupName,Config)`. However, information functions cannot be used with `init/end_per_testcase(TestCase, Config)`, as these configuration functions execute on the test case process and use the same properties as the test case (that is, the properties set by the test case information function, `TestCase()`). For a list of valid information properties and more general information, see the *Test Case Information Function*.

1.5.15 Data and Private Directories

In the data directory, `data_dir`, the test module has its own files needed for the testing. The name of `data_dir` is the the name of the test suite followed by `"_data"`. For example, `"some_path/foo_SUITE.beam"` has the

data directory `"some_path/foo_SUITE_data/"`. Use this directory for portability, that is, to avoid hardcoding directory names in your suite. As the data directory is stored in the same directory as your test suite, you can rely on its existence at runtime, even if the path to your test suite directory has changed between test suite implementation and execution.

`priv_dir` is the private directory for the test cases. This directory can be used whenever a test case (or configuration function) needs to write something to file. The name of the private directory is generated by `Common Test`, which also creates the directory.

By default, `Common Test` creates one central private directory per test run, shared by all test cases. This is not always suitable. Especially if the same test cases are executed multiple times during a test run (that is, if they belong to a test case group with property `repeat`) and there is a risk that files in the private directory get overwritten. Under these circumstances, `Common Test` can be configured to create one dedicated private directory per test case and execution instead. This is accomplished with the flag/option `create_priv_dir` (to be used with the *`ct_run`* program, the *`ct:run_test/1`* function, or as test specification term). There are three possible values for this option as follows:

- `auto_per_run`
- `auto_per_tc`
- `manual_per_tc`

The first value indicates the default `priv_dir` behavior, that is, one private directory created per test run. The two latter values tell `Common Test` to generate a unique test directory name per test case and execution. If the auto version is used, **all** private directories are created automatically. This can become very inefficient for test runs with many test cases or repetitions, or both. Therefore, if the manual version is used instead, the test case must tell `Common Test` to create `priv_dir` when it needs it. It does this by calling the function *`ct:make_priv_dir/0`*.

Note:

Do not depend on the current working directory for reading and writing data files, as this is not portable. All scratch files are to be written in the `priv_dir` and all data files are to be located in `data_dir`. Also, the `Common Test` server sets the current working directory to the test case log directory at the start of every case.

1.5.16 Execution Environment

Each test case is executed by a dedicated Erlang process. The process is spawned when the test case starts, and terminated when the test case is finished. The configuration functions `init_per_testcase` and `end_per_testcase` execute on the same process as the test case.

The configuration functions `init_per_suite` and `end_per_suite` execute, like test cases, on dedicated Erlang processes.

1.5.17 Timetrap Time-Outs

The default time limit for a test case is 30 minutes, unless a `timetrap` is specified either by the suite-, group-, or test case information function. The timetrap time-out value defined by `suite/0` is the value that is used for each test case in the suite (and for the configuration functions `init_per_suite/1`, `end_per_suite/1`, `init_per_group/2`, and `end_per_group/2`). A timetrap value defined by `group(GroupName)` overrides one defined by `suite()` and is used for each test case in group `GroupName`, and any of its subgroups. If a timetrap value is defined by `group/1` for a subgroup, it overrides that of its higher level groups. Timetrap values set by individual test cases (by the test case information function) override both group- and suite- level timetraps.

A timetrap can also be set or reset dynamically during the execution of a test case, or configuration function. This is done by calling *`ct:timetrap/1`*. This function cancels the current timetrap and starts a new one (that stays active until time-out, or end of the current function).

Timetrap values can be extended with a multiplier value specified at startup with option `multiply_timetraps`. It is also possible to let the test server decide to scale up timetrap time-out values automatically. That is, if tools such as `cover` or `trace` are running during the test. This feature is disabled by default and can be enabled with start option `scale_timetraps`.

If a test case needs to suspend itself for a time that also gets multipled by `multiply_timetraps` (and possibly also scaled up if `scale_timetraps` is enabled), the function *`ct:sleep/1`* can be used (instead of, for example, `timer:sleep/1`).

A function (`fun/0` or `{Mod,Func,Args}` (MFA) tuple) can be specified as timetrap value in the suite-, group- and test case information function, and as argument to function *`ct:timetrap/1`*.

Examples:

```
{timetrap,{my_test_utils,timetrap,[?MODULE,system_start]}}
```

```
ct:timetrap(fun() -> my_timetrap(TestCaseName, Config) end)
```

The user timetrap function can be used for two things as follows:

- To act as a timetrap. The time-out is triggered when the function returns.
- To return a timetrap time value (other than a function).

Before execution of the timetrap function (which is performed on a parallel, dedicated timetrap process), `Common Test` cancels any previously set timer for the test case or configuration function. When the timetrap function returns, the time-out is triggered, **unless** the return value is a valid timetrap time, such as an integer, or a `{SecMinOrHourTag,Time}` tuple (for details, see module *common_test*). If a time value is returned, a new timetrap is started to generate a time-out after the specified time.

The user timetrap function can return a time value after a delay. The effective timetrap time is then the delay time **plus** the returned time.

1.5.18 Logging - Categories and Verbosity Levels

`Common Test` provides the following three main functions for printing strings:

- `ct:log(Category, Importance, Format, FormatArgs, Opts)`
- `ct:print(Category, Importance, Format, FormatArgs)`
- `ct:pal(Category, Importance, Format, FormatArgs)`

The *`log/1,2,3,4,5`* function prints a string to the test case log file. The *`print/1,2,3,4`* function prints the string to screen. The *`pal/1,2,3,4`* function prints the same string both to file and screen. The functions are described in module *ct*.

The optional `Category` argument can be used to categorize the log printout. Categories can be used for two things as follows:

- To compare the importance of the printout to a specific verbosity level.
- To format the printout according to a user-specific HTML Style Sheet (CSS).

Argument `Importance` specifies a level of importance that, compared to a verbosity level (general and/or set per category), determines if the printout is to be visible. `Importance` is any integer in the range 0..99. Predefined constants exist in the `ct.hrl` header file. The default importance level, `?STD_IMPORTANCE` (used if argument `Importance` is not provided), is 50. This is also the importance used for standard I/O, for example, from printouts made with `io:format/2`, `io:put_chars/1`, and so on.

`Importance` is compared to a verbosity level set by the `verbosity` start flag/option. The level can be set per category or generally, or both. If `verbosity` is not set by the user, a level of 100 (`?MAX_VERBOSITY` = all printouts visible) is used as default value. `Common Test` performs the following test:

```
Importance >= (100-VerbosityLevel)
```

The constant `?STD_VERBOSITY` has value 50 (see `ct.hrl`). At this level, all standard I/O gets printed. If a lower verbosity level is set, standard I/O printouts are ignored. Verbosity level 0 effectively turns all logging off (except from printouts made by `Common Test` itself).

The general verbosity level is not associated with any particular category. This level sets the threshold for the standard I/O printouts, uncategorized `ct:log/print/pal` printouts, and printouts for categories with undefined verbosity level.

Examples:

Some printouts during test case execution:

```
io:format("1. Standard IO, importance = ~w~n", [?STD_IMPORTANCE]),
ct:log("2. Uncategorized, importance = ~w", [?STD_IMPORTANCE]),
ct:log(info, "3. Categorized info, importance = ~w", [?STD_IMPORTANCE]),
ct:log(info, ?LOW_IMPORTANCE, "4. Categorized info, importance = ~w", [?LOW_IMPORTANCE]),
ct:log(error, ?HI_IMPORTANCE, "5. Categorized error, importance = ~w", [?HI_IMPORTANCE]),
ct:log(error, ?MAX_IMPORTANCE, "6. Categorized error, importance = ~w", [?MAX_IMPORTANCE]),
```

If starting the test with a general verbosity level of 50 (`?STD_VERBOSITY`):

```
$ ct_run -verbosity 50
```

the following is printed:

```
1. Standard IO, importance = 50
2. Uncategorized, importance = 50
3. Categorized info, importance = 50
5. Categorized error, importance = 75
6. Categorized error, importance = 99
```

If starting the test with:

```
$ ct_run -verbosity 1 and info 75
```

the following is printed:

```
3. Categorized info, importance = 50
4. Categorized info, importance = 25
6. Categorized error, importance = 99
```

Note that the category argument is not required in order to only specify the importance of a printout. Example:

```
ct:pal(?LOW_IMPORTANCE, "Info report: ~p", [Info])
```

Or perhaps in combination with constants:

```
-define(INFO, ?LOW_IMPORTANCE).
-define(ERROR, ?HI_IMPORTANCE).

ct:log(?INFO, "Info report: ~p", [Info])
ct:pal(?ERROR, "Error report: ~p", [Error])
```

The functions *ct:set_verbosity/2* and *ct:get_verbosity/1* may be used to modify and read verbosity levels during test execution.

The arguments `Format` and `FormatArgs` in `ct:log/print/pal` are always passed on to the STDLIB function `io:format/3` (For details, see the *io* manual page).

`ct:pal/4` and `ct:log/5` add headers to strings being printed to the log file. The strings are also wrapped in div tags with a CSS class attribute, so that stylesheet formatting can be applied. To disable this feature for a printout (i.e. to get a result similar to using `io:format/2`), call `ct:log/5` with the `no_css` option.

How categories can be mapped to CSS tags is documented in section *HTML Style Sheets* in section Running Tests and Analyzing Results.

Common Test will escape special HTML characters (<, > and &) in printouts to the log file made with `ct:pal/4` and `io:format/2`. In order to print strings with HTML tags to the log, use the `ct:log/3,4,5` function. The character escaping feature is per default disabled for `ct:log/3,4,5` but can be enabled with the `esc_chars` option in the `Opts` list, see *ct:log/3,4,5*.

If the character escaping feature needs to be disabled (typically for backwards compatibility reasons), use the `ct_run` start flag `-no_esc_chars`, or the `ct:run_test/1` start option `{esc_chars,Bool}` (this start option is also supported in test specifications).

For more information about log files, see section *Log Files* in section Running Tests and Analyzing Results.

1.5.19 Illegal Dependencies

Even though it is highly efficient to write test suites with the `Common Test` framework, mistakes can be made, mainly because of illegal dependencies. Some of the more frequent mistakes from our own experience with running the Erlang/OTP test suites follows:

- Depending on current directory, and writing there:

  This is a common error in test suites. It is assumed that the current directory is the same as the author used as current directory when the test case was developed. Many test cases even try to write scratch files to this directory. Instead `data_dir` and `priv_dir` are to be used to locate data and for writing scratch files.
- Depending on execution order:

  During development of test suites, make no assumptions on the execution order of the test cases or suites. For example, a test case must not assume that a server it depends on is already started by a previous test case. Reasons for this follows:

  - The user/operator can specify the order at will, and maybe a different execution order is sometimes more relevant or efficient.
  - If the user specifies a whole directory of test suites for the test, the execution order of the suites depends on how the files are listed by the operating system, which varies between systems.
  - If a user wants to run only a subset of a test suite, there is no way one test case could successfully depend on another.
- Depending on Unix:

  Running Unix commands through `os:cmd` are likely not to work on non-Unix platforms.
- Nested test cases:

  Starting a test case from another not only tests the same thing twice, but also makes it harder to follow what is being tested. Also, if the called test case fails for some reason, so do the caller. This way, one error gives cause to several error reports, which is to be avoided.

  Functionality common for many test case functions can be implemented in common help functions. If these functions are useful for test cases across suites, put the help functions into common help modules.

- Failure to crash or exit when things go wrong:

  Making requests without checking that the return value indicates success can be OK if the test case fails later, but it is never acceptable just to print an error message (into the log file) and return successfully. Such test cases do harm, as they create a false sense of security when overviewing the test results.
- Messing up for subsequent test cases:

  Test cases are to restore as much of the execution environment as possible, so that subsequent test cases do not crash because of their execution order. The function `end_per_testcase` is suitable for this.

1.6 Test Structure

1.6.1 General

A test is performed by running one or more test suites. A test suite consists of test cases, configuration functions, and information functions. Test cases can be grouped in so called test case groups. A test suite is an Erlang module and test cases are implemented as Erlang functions. Test suites are stored in test directories.

1.6.2 Skipping Test Cases

Certain test cases can be skipped, for example, if you know beforehand that a specific test case fails. The reason can be functionality that is not yet implemented, a bug that is known but not yet fixed, or some functionality that does not work or is not applicable on a specific platform.

Test cases can be skipped in the following ways:

- Using `skip_suites` and `skip_cases` terms in *test specifications*.
- Returning `{skip,Reason}` from function `init_per_testcase/2` or `init_per_suite/1`.
- Returning `{skip,Reason}` from the execution clause of the test case. The execution clause is called, so the author must ensure that the test case does not run.

When a test case is skipped, it is noted as `SKIPPED` in the HTML log.

1.6.3 Definition of Terms

Auto-skipped test case

When a configuration function fails (that is, terminates unexpectedly), the test cases depending on the configuration function are skipped automatically by `Common Test`. The status of the test cases is then "auto-skipped". Test cases are also "auto-skipped" by `Common Test` if the required configuration data is unavailable at runtime.

Configuration function

A function in a test suite that is meant to be used for setting up, cleaning up, and/or verifying the state and environment on the System Under Test (SUT) and/or the `Common Test` host node, so that a test case (or a set of test cases) can execute correctly.

Configuration file

A file containing data related to a test and/or an SUT, for example, protocol server addresses, client login details, and hardware interface addresses. That is, any data that is to be handled as variable in the suite and not be hard-coded.

Configuration variable

A name (an Erlang atom) associated with a data value read from a configuration file.

`data_dir`

Data directory for a test suite. This directory contains any files used by the test suite, for example, extra Erlang modules, binaries, or data files.

Information function

A function in a test suite that returns a list of properties (read by the `Common Test` server) that describes the conditions for executing the test cases in the suite.

Major log file

An overview and summary log file for one or more test suites.

Minor log file

A log file for one particular test case. Also called the test case log file.

`priv_dir`

Private directory for a test suite. This directory is to be used when the test suite needs to write to files.

`ct_run`

The name of an executable program that can be used as an interface for specifying and running tests with `Common Test`.

Test case

A single test included in a test suite. A test case is implemented as a function in a test suite module.

Test case group

A set of test cases sharing configuration functions and execution properties. The execution properties specify if the test cases in the group are to be executed in random order, in parallel, or in sequence, and if the execution of the group is be repeated. Test case groups can also be nested. That is, a group can, besides test cases, contain subgroups.

Test suite

An Erlang module containing a collection of test cases for a specific functional area.

Test directory

A directory containing one or more test suite modules, that is, a group of test suites.

Argument `Config`

A list of key-value tuples (that is, a property list) containing runtime configuration data passed from the configuration functions to the test cases.

User-skipped test case

The status of a test case explicitly skipped in any of the ways described in section *Skipping Test Cases*.

1.7 Examples and Templates

1.7.1 Test Suite Example

The following example test suite shows some tests of a database server:

```
-module(db_data_type_SUITE).

-include_lib("common_test/include/ct.hrl").

%% Test server callbacks
-export([suite/0, all/0,
  init_per_suite/1, end_per_suite/1,
  init_per_testcase/2, end_per_testcase/2]).

%% Test cases
-export([string/1, integer/1]).

-define(CONNECT_STR, "DSN=sqlserver;UID=alladin;PWD=sesame").

%%--------------------------------------------------------------------
%% COMMON TEST CALLBACK FUNCTIONS
%%--------------------------------------------------------------------

%%--------------------------------------------------------------------
%% Function: suite() -> Info
%%
%% Info = [tuple()]
%%   List of key/value pairs.
%%
%% Description: Returns list of tuples to set default properties
%%              for the suite.
%%--------------------------------------------------------------------
suite() ->
    [{timetrap,{minutes,1}}].

%%--------------------------------------------------------------------
%% Function: init_per_suite(Config0) -> Config1
%%
%% Config0 = Config1 = [tuple()]
%%   A list of key/value pairs, holding the test case configuration.
%%
%% Description: Initialization before the suite.
%%--------------------------------------------------------------------
init_per_suite(Config) ->
    {ok, Ref} = db:connect(?CONNECT_STR, []),
    TableName = db_lib:unique_table_name(),
    [{con_ref, Ref },{table_name, TableName}| Config].

%%--------------------------------------------------------------------
%% Function: end_per_suite(Config) -> term()
%%
%% Config = [tuple()]
%%   A list of key/value pairs, holding the test case configuration.
%%
%% Description: Cleanup after the suite.
%%--------------------------------------------------------------------
end_per_suite(Config) ->
    Ref = ?config(con_ref, Config),
    db:disconnect(Ref),
    ok.

%%--------------------------------------------------------------------
%% Function: init_per_testcase(TestCase, Config0) -> Config1
%%
%% TestCase = atom()
%%   Name of the test case that is about to run.
%% Config0 = Config1 = [tuple()]
%%   A list of key/value pairs, holding the test case configuration.
%%
%% Description: Initialization before each test case.
```

```
%%--------------------------------------------------------------------
init_per_testcase(Case, Config) ->
    Ref = ?config(con_ref, Config),
    TableName = ?config(table_name, Config),
    ok = db:create_table(Ref, TableName, table_type(Case)),
    Config.

%%--------------------------------------------------------------------
%% Function: end_per_testcase(TestCase, Config) -> term()
%%
%% TestCase = atom()
%%   Name of the test case that is finished.
%% Config = [tuple()]
%%   A list of key/value pairs, holding the test case configuration.
%%
%% Description: Cleanup after each test case.
%%--------------------------------------------------------------------
end_per_testcase(_Case, Config) ->
    Ref = ?config(con_ref, Config),
    TableName = ?config(table_name, Config),
    ok = db:delete_table(Ref, TableName),
    ok.

%%--------------------------------------------------------------------
%% Function: all() -> GroupsAndTestCases
%%
%% GroupsAndTestCases = [{group,GroupName} | TestCase]
%% GroupName = atom()
%%   Name of a test case group.
%% TestCase = atom()
%%   Name of a test case.
%%
%% Description: Returns the list of groups and test cases that
%%              are to be executed.
%%--------------------------------------------------------------------
all() ->
    [string, integer].


%%--------------------------------------------------------------------
%% TEST CASES
%%--------------------------------------------------------------------

string(Config) ->
    insert_and_lookup(dummy_key, "Dummy string", Config).

integer(Config) ->
    insert_and_lookup(dummy_key, 42, Config).


insert_and_lookup(Key, Value, Config) ->
    Ref = ?config(con_ref, Config),
    TableName = ?config(table_name, Config),
    ok = db:insert(Ref, TableName, Key, Value),
    [Value] = db:lookup(Ref, TableName, Key),
    ok = db:delete(Ref, TableName, Key),
    [] = db:lookup(Ref, TableName, Key),
    ok.
```

1.7.2 Test Suite Templates

The Erlang mode for the Emacs editor includes two `Common Test` test suite templates, one with extensive information in the function headers, and one with minimal information. A test suite template provides a quick start for

implementing a suite from scratch and gives a good overview of the available callback functions. The two templates follows:

Large Common Test Suite

```
%%%-----------------------------------------------------------------
%%% File    : example_SUITE.erl
%%% Author  :
%%% Description :
%%%
%%% Created :
%%%-----------------------------------------------------------------
-module(example_SUITE).

%% Note: This directive should only be used in test suites.
-compile(export_all).

-include_lib("common_test/include/ct.hrl").

%%--------------------------------------------------------------------
%% COMMON TEST CALLBACK FUNCTIONS
%%--------------------------------------------------------------------

%%--------------------------------------------------------------------
%% Function: suite() -> Info
%%
%% Info = [tuple()]
%%   List of key/value pairs.
%%
%% Description: Returns list of tuples to set default properties
%%              for the suite.
%%
%% Note: The suite/0 function is only meant to be used to return
%% default data values, not perform any other operations.
%%--------------------------------------------------------------------
suite() ->
    [{timetrap,{minutes,10}}].

%%--------------------------------------------------------------------
%% Function: init_per_suite(Config0) ->
%%               Config1 | {skip,Reason} | {skip_and_save,Reason,Config1}
%%
%% Config0 = Config1 = [tuple()]
%%   A list of key/value pairs, holding the test case configuration.
%% Reason = term()
%%   The reason for skipping the suite.
%%
%% Description: Initialization before the suite.
%%
%% Note: This function is free to add any key/value pairs to the Config
%% variable, but should NOT alter/remove any existing entries.
%%--------------------------------------------------------------------
init_per_suite(Config) ->
    Config.

%%--------------------------------------------------------------------
%% Function: end_per_suite(Config0) -> term() | {save_config,Config1}
%%
%% Config0 = Config1 = [tuple()]
%%   A list of key/value pairs, holding the test case configuration.
%%
%% Description: Cleanup after the suite.
%%--------------------------------------------------------------------
end_per_suite(_Config) ->
    ok.

%%--------------------------------------------------------------------
%% Function: init_per_group(GroupName, Config0) ->
%%               Config1 | {skip,Reason} | {skip_and_save,Reason,Config1}
%%
```

```
%% GroupName = atom()
%%   Name of the test case group that is about to run.
%% Config0 = Config1 = [tuple()]
%%   A list of key/value pairs, holding configuration data for the group.
%% Reason = term()
%%   The reason for skipping all test cases and subgroups in the group.
%%
%% Description: Initialization before each test case group.
%%--------------------------------------------------------------------
init_per_group(_GroupName, Config) ->
    Config.

%%--------------------------------------------------------------------
%% Function: end_per_group(GroupName, Config0) ->
%%               term() | {save_config,Config1}
%%
%% GroupName = atom()
%%   Name of the test case group that is finished.
%% Config0 = Config1 = [tuple()]
%%   A list of key/value pairs, holding configuration data for the group.
%%
%% Description: Cleanup after each test case group.
%%--------------------------------------------------------------------
end_per_group(_GroupName, _Config) ->
    ok.

%%--------------------------------------------------------------------
%% Function: init_per_testcase(TestCase, Config0) ->
%%               Config1 | {skip,Reason} | {skip_and_save,Reason,Config1}
%%
%% TestCase = atom()
%%   Name of the test case that is about to run.
%% Config0 = Config1 = [tuple()]
%%   A list of key/value pairs, holding the test case configuration.
%% Reason = term()
%%   The reason for skipping the test case.
%%
%% Description: Initialization before each test case.
%%
%% Note: This function is free to add any key/value pairs to the Config
%% variable, but should NOT alter/remove any existing entries.
%%--------------------------------------------------------------------
init_per_testcase(_TestCase, Config) ->
    Config.

%%--------------------------------------------------------------------
%% Function: end_per_testcase(TestCase, Config0) ->
%%               term() | {save_config,Config1} | {fail,Reason}
%%
%% TestCase = atom()
%%   Name of the test case that is finished.
%% Config0 = Config1 = [tuple()]
%%   A list of key/value pairs, holding the test case configuration.
%% Reason = term()
%%   The reason for failing the test case.
%%
%% Description: Cleanup after each test case.
%%--------------------------------------------------------------------
end_per_testcase(_TestCase, _Config) ->
    ok.

%%--------------------------------------------------------------------
%% Function: groups() -> [Group]
%%
%% Group = {GroupName,Properties,GroupsAndTestCases}
```

```
%% GroupName = atom()
%%   The name of the group.
%% Properties = [parallel | sequence | Shuffle | {RepeatType,N}]
%%   Group properties that may be combined.
%% GroupsAndTestCases = [Group | {group,GroupName} | TestCase]
%% TestCase = atom()
%%   The name of a test case.
%% Shuffle = shuffle | {shuffle,Seed}
%%   To get cases executed in random order.
%% Seed = {integer(),integer(),integer()}
%% RepeatType = repeat | repeat_until_all_ok | repeat_until_all_fail |
%%              repeat_until_any_ok | repeat_until_any_fail
%%   To get execution of cases repeated.
%% N = integer() | forever
%%
%% Description: Returns a list of test case group definitions.
%%--------------------------------------------------------------------
groups() ->
    [].

%%--------------------------------------------------------------------
%% Function: all() -> GroupsAndTestCases | {skip,Reason}
%%
%% GroupsAndTestCases = [{group,GroupName} | TestCase]
%% GroupName = atom()
%%   Name of a test case group.
%% TestCase = atom()
%%   Name of a test case.
%% Reason = term()
%%   The reason for skipping all groups and test cases.
%%
%% Description: Returns the list of groups and test cases that
%%              are to be executed.
%%--------------------------------------------------------------------
all() ->
    [my_test_case].


%%--------------------------------------------------------------------
%% TEST CASES
%%--------------------------------------------------------------------

%%--------------------------------------------------------------------
%% Function: TestCase() -> Info
%%
%% Info = [tuple()]
%%   List of key/value pairs.
%%
%% Description: Test case info function - returns list of tuples to set
%%              properties for the test case.
%%
%% Note: This function is only meant to be used to return a list of
%% values, not perform any other operations.
%%--------------------------------------------------------------------
my_test_case() ->
    [].

%%--------------------------------------------------------------------
%% Function: TestCase(Config0) ->
%%               ok | exit() | {skip,Reason} | {comment,Comment} |
%%               {save_config,Config1} | {skip_and_save,Reason,Config1}
%%
%% Config0 = Config1 = [tuple()]
%%   A list of key/value pairs, holding the test case configuration.
%% Reason = term()
```

```
%%   The reason for skipping the test case.
%% Comment = term()
%%   A comment about the test case that will be printed in the html log.
%%
%% Description: Test case function. (The name of it must be specified in
%%              the all/0 list or in a test case group for the test case
%%              to be executed).
%%--------------------------------------------------------------------
my_test_case(_Config) ->
    ok.
```

Small Common Test Suite

```
%%%-----------------------------------------------------------------
%%% File    : example_SUITE.erl
%%% Author  :
%%% Description :
%%%
%%% Created :
%%%-----------------------------------------------------------------
-module(example_SUITE).

-compile(export_all).

-include_lib("common_test/include/ct.hrl").

%%--------------------------------------------------------------------
%% Function: suite() -> Info
%% Info = [tuple()]
%%--------------------------------------------------------------------
suite() ->
    [{timetrap,{seconds,30}}].

%%--------------------------------------------------------------------
%% Function: init_per_suite(Config0) ->
%%               Config1 | {skip,Reason} | {skip_and_save,Reason,Config1}
%% Config0 = Config1 = [tuple()]
%% Reason = term()
%%--------------------------------------------------------------------
init_per_suite(Config) ->
    Config.

%%--------------------------------------------------------------------
%% Function: end_per_suite(Config0) -> term() | {save_config,Config1}
%% Config0 = Config1 = [tuple()]
%%--------------------------------------------------------------------
end_per_suite(_Config) ->
    ok.

%%--------------------------------------------------------------------
%% Function: init_per_group(GroupName, Config0) ->
%%               Config1 | {skip,Reason} | {skip_and_save,Reason,Config1}
%% GroupName = atom()
%% Config0 = Config1 = [tuple()]
%% Reason = term()
%%--------------------------------------------------------------------
init_per_group(_GroupName, Config) ->
    Config.

%%--------------------------------------------------------------------
%% Function: end_per_group(GroupName, Config0) ->
%%               term() | {save_config,Config1}
%% GroupName = atom()
%% Config0 = Config1 = [tuple()]
%%--------------------------------------------------------------------
end_per_group(_GroupName, _Config) ->
    ok.

%%--------------------------------------------------------------------
%% Function: init_per_testcase(TestCase, Config0) ->
%%               Config1 | {skip,Reason} | {skip_and_save,Reason,Config1}
%% TestCase = atom()
%% Config0 = Config1 = [tuple()]
%% Reason = term()
%%--------------------------------------------------------------------
init_per_testcase(_TestCase, Config) ->
    Config.
```

```
%%--------------------------------------------------------------------
%% Function: end_per_testcase(TestCase, Config0) ->
%%               term() | {save_config,Config1} | {fail,Reason}
%% TestCase = atom()
%% Config0 = Config1 = [tuple()]
%% Reason = term()
%%--------------------------------------------------------------------
end_per_testcase(_TestCase, _Config) ->
    ok.

%%--------------------------------------------------------------------
%% Function: groups() -> [Group]
%% Group = {GroupName,Properties,GroupsAndTestCases}
%% GroupName = atom()
%% Properties = [parallel | sequence | Shuffle | {RepeatType,N}]
%% GroupsAndTestCases = [Group | {group,GroupName} | TestCase]
%% TestCase = atom()
%% Shuffle = shuffle | {shuffle,{integer(),integer(),integer()}}
%% RepeatType = repeat | repeat_until_all_ok | repeat_until_all_fail |
%%              repeat_until_any_ok | repeat_until_any_fail
%% N = integer() | forever
%%--------------------------------------------------------------------
groups() ->
    [].

%%--------------------------------------------------------------------
%% Function: all() -> GroupsAndTestCases | {skip,Reason}
%% GroupsAndTestCases = [{group,GroupName} | TestCase]
%% GroupName = atom()
%% TestCase = atom()
%% Reason = term()
%%--------------------------------------------------------------------
all() ->
    [my_test_case].

%%--------------------------------------------------------------------
%% Function: TestCase() -> Info
%% Info = [tuple()]
%%--------------------------------------------------------------------
my_test_case() ->
    [].

%%--------------------------------------------------------------------
%% Function: TestCase(Config0) ->
%%               ok | exit() | {skip,Reason} | {comment,Comment} |
%%               {save_config,Config1} | {skip_and_save,Reason,Config1}
%% Config0 = Config1 = [tuple()]
%% Reason = term()
%% Comment = term()
%%--------------------------------------------------------------------
my_test_case(_Config) ->
    ok.
```

1.8 Running Tests and Analyzing Results

1.8.1 Using the Common Test Framework

The `Common Test` framework provides a high-level operator interface for testing, providing the following features:

- Automatic compilation of test suites (and help modules)
- Creation of extra HTML pages for improved overview.
- Single-command interface for running all available tests

- Handling of configuration files specifying data related to the System Under Test (SUT) (and any other variable data)
- Mode for running multiple independent test sessions in parallel with central control and configuration

1.8.2 Automatic Compilation of Test Suites and Help Modules

When `Common Test` starts, it automatically attempts to compile any suites included in the specified tests. If particular suites are specified, only those suites are compiled. If a particular test object directory is specified (meaning all suites in this directory are to be part of the test), `Common Test` runs function `make:all/1` in the directory to compile the suites.

If compilation fails for one or more suites, the compilation errors are printed to tty and the operator is asked if the test run is to proceed without the missing suites, or be aborted. If the operator chooses to proceed, the tests having missing suites are noted in the HTML log. If `Common Test` is unable to prompt the user after compilation failure (if `Common Test` does not control `stdin`), the test run proceeds automatically without the missing suites. This behavior can however be modified with the `ct_run` flag `-abort_if_missing_suites`, or the *`ct:run_test/1`* option `{abort_if_missing_suites,TrueOrFalse}`. If `abort_if_missing_suites` is set to `true`, the test run stops immediately if some suites fail to compile.

Any help module (that is, regular Erlang module with name not ending with "_SUITE") that resides in the same test object directory as a suite, which is part of the test, is also automatically compiled. A help module is not mistaken for a test suite (unless it has a "_SUITE" name). All help modules in a particular test object directory are compiled, no matter if all or only particular suites in the directory are part of the test.

If test suites or help modules include header files stored in other locations than the test directory, these include directories can be specified by using flag `-include` with *`ct_run`*, or option `include` with `ct:run_test/1`. Also, an include path can be specified with an OS environment variable, `CT_INCLUDE_PATH`.

Example (bash):

```
$ export CT_INCLUDE_PATH=~testuser/common_suite_files/include:~testuser/
common_lib_files/include
```

`Common Test` passes all include directories (specified either with flag/option `include`, or variable `CT_INCLUDE_PATH` , or both, to the compiler.

Include directories can also be specified in test specifications, see *Test Specifications*.

If the user wants to run all test suites for a test object (or an OTP application) by specifying only the top directory (for example, with start flag/option `dir`), `Common Test` primarily looks for test suite modules in a subdirectory named `test`. If this subdirectory does not exist, the specified top directory is assumed to be the test directory, and test suites are read from there instead.

To disable the automatic compilation feature, use flag `-no_auto_compile` with `ct_run`, or option `{auto_compile,false}` with `ct:run_test/1`. With automatic compilation disabled, the user is responsible for compiling the test suite modules (and any help modules) before the test run. If the modules cannot be loaded from the local file system during startup of `Common Test`, the user must preload the modules before starting the test. `Common Test` only verifies that the specified test suites exist (that is, that they are, or can be, loaded). This is useful, for example, if the test suites are transferred and loaded as binaries through RPC from a remote node.

1.8.3 Running Tests from the OS Command Line

The *`ct_run`* program can be used for running tests from the OS command line, for example, as follows:

- `ct_run -config <configfilenames> -dir <dirs>`
- `ct_run -config <configfilenames> -suite <suiteswithfullpath>`
- `ct_run -userconfig <callbackmodulename> <configfilenames> -suite <suiteswithfullpath>`

- `ct_run -config <configfilenames> -suite <suitewithfullpath> -group <groups> -case <casenames>`

Examples:

```
$ ct_run -config $CFGS/sys1.cfg $CFGS/sys2.cfg -dir $SYS1_TEST $SYS2_TEST
$ ct_run -userconfig ct_config_xml $CFGS/sys1.xml $CFGS/sys2.xml -dir $SYS1_TEST $SYS2_TEST
$ ct_run -suite $SYS1_TEST/setup_SUITE $SYS2_TEST/config_SUITE
$ ct_run -suite $SYS1_TEST/setup_SUITE -case start stop
$ ct_run -suite $SYS1_TEST/setup_SUITE -group installation -case start stop
```

The flags `dir`, `suite`, and `group/case` can be combined. For example, to run `x_SUITE` and `y_SUITE` in directory `testdir`, as follows:

```
$ ct_run -dir ./testdir -suite x_SUITE y_SUITE
```

This has the same effect as the following:

```
$ ct_run -suite ./testdir/x_SUITE ./testdir/y_SUITE
```

For details, see *Test Case Group Execution*.

The following flags can also be used with *`ct_run`*:

`-help`

Lists all available start flags.

`-logdir <dir>`

Specifies where the HTML log files are to be written.

`-label <name_of_test_run>`

Associates the test run with a name that gets printed in the overview HTML log files.

`-refresh_logs`

Refreshes the top-level HTML index files.

`-shell`

Starts interactive shell mode (described later).

`-step [step_opts]`

Steps through test cases using the Erlang Debugger (described later).

`-spec <testspecs>`

Uses test specification as input (described later).

`-allow_user_terms`

Allows user-specific terms in a test specification (described later).

`-silent_connections [conn_types]`

, tells `Common Test` to suppress printouts for specified connections (described later).

`-stylesheet <css_file>`

Points out a user HTML style sheet (described later).

`-cover <cover_cfg_file>`

To perform code coverage test (see *Code Coverage Analysis*).

`-cover_stop <bool>`

To specify if the `cover` tool is to be stopped after the test is completed (see *Code Coverage Analysis*).

`-event_handler <event_handlers>`

To install *event handlers*.

`-event_handler_init <event_handlers>`

To install *event handlers* including start arguments.

`-ct_hooks <ct_hooks>`

To install *Common Test Hooks* including start arguments.

`-enable_builtin_hooks <bool>`

To enable or disable *Built-in Common Test Hooks*. Default is `true`.

`-include`

Specifies include directories (described earlier).

`-no_auto_compile`

Disables the automatic test suite compilation feature (described earlier).

`-abort_if_missing_suites`

Aborts the test run if one or more suites fail to compile (described earlier).

`-multiply_timetraps <n>`

Extends *timetrap time-out* values.

`-scale_timetraps <bool>`

Enables automatic *timetrap time-out* scaling.

`-repeat <n>`

Tells `Common Test` to repeat the tests `n` times (described later).

`-duration <time>`

Tells `Common Test` to repeat the tests for duration of time (described later).

`-until <stop_time>`

Tells `Common Test` to repeat the tests until `stop_time` (described later).

`-force_stop [skip_rest]`

On time-out, the test run is aborted when the current test job is finished. If `skip_rest` is provided, the remaining test cases in the current test job are skipped (described later).

`-decrypt_key <key>`

Provides a decryption key for *encrypted configuration files*.

`-decrypt_file <key_file>`

Points out a file containing a decryption key for *encrypted configuration files*.

`-basic_html`

Switches off HTML enhancements that can be incompatible with older browsers.

`-logopts <opts>`

Enables modification of the logging behavior, see *Log options*.

`-verbosity <levels>`

Sets *verbosity levels for printouts*.

`-no_esc_chars`

Disables automatic escaping of special HTML characters. See the *Logging chapter*.

Note:

Directories passed to `Common Test` can have either relative or absolute paths.

Note:

Any start flags to the Erlang runtime system (application ERTS) can also be passed as parameters to `ct_run`. It is, for example, useful to be able to pass directories to be added to the Erlang code server search path with flag `-pa` or `-pz`. If you have common help- or library modules for test suites (separately compiled), stored in other directories than the test suite directories, these `help/lib` directories are preferably added to the code path this way.

Example:

```
$ ct_run -dir ./chat_server -logdir ./chat_server/testlogs -pa $PWD/chat_server/ebin
```

The absolute path of directory `chat_server/ebin` is here passed to the code server. This is essential because relative paths are stored by the code server as relative, and `Common Test` changes the current working directory of ERTS during the test run.

The `ct_run` program sets the exit status before shutting down. The following values are defined:

- `0` indicates a successful testrun, that is, without failed or auto-skipped test cases.
- `1` indicates that one or more test cases have failed, or have been auto-skipped.
- `2` indicates that the test execution has failed because of, for example, compilation errors, or an illegal return value from an information function.

If auto-skipped test cases do not affect the exit status. The default behavior can be changed using start flag:

```
-exit_status ignore_config
```

Note:

Executing `ct_run` without start flags is equal to the command: `ct_run -dir ./`

For more information about the `ct_run` program, see module *`ct_run`* and section *Installation*.

1.8.4 Running Tests from the Erlang Shell or from an Erlang Program

`Common Test` provides an Erlang API for running tests. The main (and most flexible) function for specifying and executing tests is *`ct:run_test/1`*. It takes the same start parameters as *`ct_run`*, but the flags are instead specified as options in a list of key-value tuples. For example, a test specified with `ct_run` as follows:

```
$ ct_run -suite ./my_SUITE -logdir ./results
```

is with *`ct:run_test/1`* specified as:

```
1> ct:run_test([{suite,"./my_SUITE"},{logdir,"./results"}]).
```

The function returns the test result, represented by the tuple `{Ok,Failed,{UserSkipped,AutoSkipped}}`, where each element is an integer. If test execution fails, the function returns the tuple `{error,Reason}`, where the term `Reason` explains the failure.

The default start option `{dir,Cwd}` (to run all suites in the current working directory) is used if the function is called with an empty list of options.

Releasing the Erlang Shell

During execution of tests started with *`ct:run_test/1`*, the Erlang shell process, controlling `stdin`, remains the top-level process of the `Common Test` system of processes. Consequently, the Erlang shell is not available for interaction during the test run. If this is not desirable, for example, because the shell is needed for debugging purposes or for interaction with the SUT during test execution, set start option `release_shell` to `true` (in the call to `ct:run_test/1` or by using the corresponding test specification term, described later). This makes `Common Test` release the shell immediately after the test suite compilation stage. To accomplish this, a test runner process is spawned to take control of the test execution. The effect is that `ct:run_test/1` returns the pid of this process rather than the test result, which instead is printed to tty at the end of the test run.

Note:

To use the functions *`ct:break/1,2`* and *`ct:continue/0,1`*, `release_shell` **must** be set to `true`.

For details, see *`ct:run_test/1`* manual page.

1.8.5 Test Case Group Execution

With the `ct_run` flag, or `ct:run_test/1` option `group`, one or more test case groups can be specified, optionally in combination with specific test cases. The syntax for specifying groups on the command line is as follows:

```
$ ct_run -group <group_names_or_paths> [-case <cases>]
```

The syntax in the Erlang shell is as follows:

```
1> ct:run_test([{group,GroupsNamesOrPaths}, {case,Cases}]).
```

Parameter `group_names_or_paths` specifies one or more group names and/or one or more group paths. At startup, `Common Test` searches for matching groups in the group definitions tree (that is, the list returned from `Suite:groups/0`; for details, see section *Test Case Groups*.

Given a group name, say `g`, `Common Test` searches for all paths leading to `g`. By path is meant a sequence of nested groups, which must be followed to get from the top-level group to `g`. To execute the test cases in group `g`, `Common Test` must call the `init_per_group/2` function for each group in the path to `g`, and all corresponding `end_per_group/2` functions afterwards. This is because the configuration of a test case in `g` (and its `Config` input data) depends on `init_per_testcase(TestCase, Config)` and its return value, which in turn depends on `init_per_group(g, Config)` and its return value, which in turn depends on `init_per_group/2` of the group above `g`, and so on, all the way up to the top-level group.

This means that if there is more than one way to locate a group (and its test cases) in a path, the result of the group search operation is a number of tests, all of which are to be performed. `Common Test` interprets a group specification that consists of a single name as follows:

"Search and find all paths in the group definitions tree that lead to the specified group and, for each path, create a test that does the following, in order:

- Executes all configuration functions in the path to the specified group.
- Executes all, or all matching, test cases in this group.
- Executes all, or all matching, test cases in all subgroups of the group."

The user can specify a specific group path with parameter `group_names_or_paths`. With this type of specification execution of unwanted groups (in otherwise matching paths), and/or the execution of subgroups can be avoided. The command line syntax of the group path is a list of group names in the path, for example:

```
$ ct_run -suite "./x_SUITE" -group [g1,g3,g4] -case tc1 tc5
```

The syntax in the Erlang shell is as follows (requires a list within the groups list):

```
1>    ct:run_test([{suite,"./x_SUITE"},    {group,[[g1,g3,g4]]},    {testcase,
[tc1,tc5]}]).
```

The last group in the specified path is the terminating group in the test, that is, no subgroups following this group are executed. In the previous example, `g4` is the terminating group. Hence, `Common Test` executes a test that calls all `init` configuration functions in the path to `g4`, that is, `g1..g3..g4`. It then calls test cases `tc1` and `tc5` in `g4`, and finally all `end` configuration functions in order `g4..g3..g1`.

Note:

The group path specification does not necessarily have to include **all** groups in the path to the terminating group. `Common Test` searches for all matching paths if an incomplete group path is specified.

Note:

Group names and group paths can be combined with parameter `group_names_or_paths`. Each element is treated as an individual specification in combination with parameter `cases`. The following examples illustrates this.

Examples:

```
-module(x_SUITE).
...
%% The group definitions:
groups() ->
  [{top1,[],[tc11,tc12,
      {sub11,[],[tc12,tc13]},
      {sub12,[],[tc14,tc15,
   {sub121,[],[tc12,tc16]}]}]},

   {top2,[],[{group,sub21},{group,sub22}]},
   {sub21,[],[tc21,{group,sub2X2}]},
   {sub22,[],[{group,sub221},tc21,tc22,{group,sub2X2}]},
   {sub221,[],[tc21,tc23]},
   {sub2X2,[],[tc21,tc24]}].
```

The following executes two tests, one for all cases and all subgroups under `top1`, and one for all under `top2`:

```
$ ct_run -suite "x_SUITE" -group all
1> ct:run_test([{suite,"x_SUITE"}, {group,all}]).
```

Using `-group top1 top2`, or `{group,[top1,top2]}` gives the same result.

The following executes one test for all cases and subgroups under `top1`:

```
$ ct_run -suite "x_SUITE" -group top1
1> ct:run_test([{suite,"x_SUITE"}, {group,[top1]}]).
```

The following runs a test executing `tc12` in `top1` and any subgroup under `top1` where it can be found (`sub11` and `sub121`):

```
$ ct_run -suite "x_SUITE" -group top1 -case tc12
1> ct:run_test([{suite,"x_SUITE"}, {group,[top1]}, {testcase,[tc12]}]).
```

The following executes `tc12` **only** in group `top1`:

```
$ ct_run -suite "x_SUITE" -group [top1] -case tc12
1> ct:run_test([{suite,"x_SUITE"}, {group,[[top1]]}, {testcase,[tc12]}]).
```

The following searches `top1` and all its subgroups for `tc16` resulting in that this test case executes in group `sub121`:

```
$ ct_run -suite "x_SUITE" -group top1 -case tc16
1> ct:run_test([{suite,"x_SUITE"}, {group,[top1]}, {testcase,[tc16]}]).
```

Using the specific path `-group [sub121]` or `{group,[[sub121]]}` gives the same result in this example.

The following executes two tests, one including all cases and subgroups under `sub12`, and one with **only** the test cases in `sub12`:

```
$ ct_run -suite "x_SUITE" -group sub12 [sub12]
1> ct:run_test([{suite,"x_SUITE"}, {group,[sub12,[sub12]]}]).
```

In the following example, `Common Test` finds and executes two tests, one for the path from `top2` to `sub2X2` through `sub21`, and one from `top2` to `sub2X2` through `sub22`:

```
$ ct_run -suite "x_SUITE" -group sub2X2
1> ct:run_test([{suite,"x_SUITE"}, {group,[sub2X2]}]).
```

In the following example, by specifying the unique path `top2 -> sub21 -> sub2X2`, only one test is executed. The second possible path, from `top2` to `sub2X2` (from the former example) is discarded:

```
$ ct_run -suite "x_SUITE" -group [sub21,sub2X2]
1> ct:run_test([{suite,"x_SUITE"}, {group,[[sub21,sub2X2]]}]).
```

The following executes only the test cases for `sub22` and in reverse order compared to the group definition:

```
$ ct_run -suite "x_SUITE" -group [sub22] -case tc22 tc21
1> ct:run_test([{suite,"x_SUITE"}, {group,[[sub22]]}, {testcase,[tc22,tc21]}]).
```

If a test case belonging to a group (according to the group definition) is executed without a group specification, that is, simply by (using the command line):

```
$ ct_run -suite "my_SUITE" -case my_tc
```

or (using the Erlang shell):

```
1> ct:run_test([{suite,"my_SUITE"}, {testcase,my_tc}]).
```

then `Common Test` ignores the group definition and executes the test case in the scope of the test suite only (no group configuration functions are called).

The group specification feature, as presented in this section, can also be used in *Test Specifications* (with some extra features added).

1.8.6 Running the Interactive Shell Mode

You can start `Common Test` in an interactive shell mode where no automatic testing is performed. Instead, `Common Test` starts its utility processes, installs configuration data (if any), and waits for the user to call functions (typically test case support functions) from the Erlang shell.

The shell mode is useful, for example, for debugging test suites, analyzing and debugging the SUT during "simulated" test case execution, and trying out various operations during test suite development.

To start the interactive shell mode, start an Erlang shell manually and call *`ct:install/1`* to install any configuration data you might need (use `[]` as argument otherwise). Then call *`ct:start_interactive/0`* to start `Common Test`.

If you use the `ct_run` program, you can start the Erlang shell and `Common Test` in one go by using the flag `-shell` and, optionally, flag `-config` and/or `-userconfig`.

Examples:

- `ct_run -shell`
- `ct_run -shell -config cfg/db.cfg`
- `ct_run -shell -userconfig db_login testuser x523qZ`

If no configuration file is specified with command `ct_run`, a warning is displayed. If `Common Test` has been run from the same directory earlier, the same configuration file(s) are used again. If `Common Test` has not been run from this directory before, no configuration files are available.

If any functions using "required configuration data" (for example, functions `ct_telnet` or `ct_ftp`) are to be called from the Erlang shell, first require configuration data with *`ct:require/1,2`*. This is equivalent to a `require` statement in the *Test Suite Information Function* or in the *Test Case Information Function*.

Example:

```
1> ct:require(unix_telnet, unix).
ok
2> ct_telnet:open(unix_telnet).
{ok,<0.105.0>}
4> ct_telnet:cmd(unix_telnet, "ls .").
{ok,["ls .","file1  ...",...]}
```

Everything that `Common Test` normally prints in the test case logs, are in the interactive mode written to a log named `ctlog.html` in directory `ct_run.<timestamp>`. A link to this file is available in the file named `last_interactive.html` in the directory from which you execute `ct_run`. Specifying a different root directory for the logs than the current working directory is not supported.

If you wish to exit the interactive mode (for example, to start an automated test run with *`ct:run_test/1`*), call function *`ct:stop_interactive/0`*. This shuts down the running `ct` application. Associations between configuration names and data created with `require` are consequently deleted. Function *`ct:start_interactive/0`* takes you back into interactive mode, but the previous state is not restored.

1.8.7 Step-by-Step Execution of Test Cases with the Erlang Debugger

Using `ct_run -step [opts]`, or by passing option `{step,Opts}` to *`ct:run_test/1`*, the following is possible:

- Get the Erlang Debugger started automatically.
- Use its graphical interface to investigate the state of the current test case.

- Execute the test case step-by-step and/or set execution breakpoints.

If no extra options are specified with flag/option `step`, breakpoints are set automatically on the test cases that are to be executed by `Common Test`, and those functions only. If step option `config` is specified, breakpoints are also initially set on the configuration functions in the suite, that is, `init_per_suite/1`, `end_per_suite/1`, `init_per_group/2`, `end_per_group/2`, `init_per_testcase/2` and `end_per_testcase/2`.

`Common Test` enables the Debugger auto-attach feature, which means that for every new interpreted test case function that starts to execute, a new trace window automatically pops up (as each test case executes on a dedicated Erlang process). Whenever a new test case starts, `Common Test` attempts to close the inactive trace window of the previous test case. However, if you prefer `Common Test` to leave inactive trace windows, use option `keep_inactive`.

The step functionality can be used together with flag/option `suite` and `suite` + `case/testcase`, but not together with `dir`.

1.8.8 Test Specifications

General Description

The most flexible way to specify what to test, is to use a test specification, which is a sequence of Erlang terms. The terms are normally declared in one or more text files (see *`ct:run_test/1`*), but can also be passed to `Common Test` on the form of a list (see *`ct:run_testspec/1`*). There are two general types of terms: configuration terms and test specification terms.

With configuration terms it is, for example, possible to do the following:

- Label the test run (similar to `ct_run -label`).
- Evaluate any expressions before starting the test.
- Import configuration data (similar to `ct_run -config/-userconfig`).
- Specify the top-level HTML log directory (similar to `ct_run -logdir`).
- Enable code coverage analysis (similar to `ct_run -cover`).
- Install `Common Test Hooks` (similar to `ct_run -ch_hooks`).
- Install `event_handler` plugins (similar to `ct_run -event_handler`).
- Specify include directories to be passed to the compiler for automatic compilation (similar to `ct_run -include`).
- Disable the auto-compilation feature (similar to `ct_run -no_auto_compile`).
- Set verbosity levels (similar to `ct_run -verbosity`).

Configuration terms can be combined with `ct_run` start flags or `ct:run_test/1` options. The result is, for some flags/options and terms, that the values are merged (for example, configuration files, include directories, verbosity levels, and silent connections) and for others that the start flags/options override the test specification terms (for example, log directory, label, style sheet, and auto-compilation).

With test specification terms, it is possible to state exactly which tests to run and in which order. A test term specifies either one or more suites, one or more test case groups (possibly nested), or one or more test cases in a group (or in multiple groups) or in a suite.

Any number of test terms can be declared in sequence. `Common Test` compiles by default the terms into one or more tests to be performed in one resulting test run. A term that specifies a set of test cases "swallows" one that only specifies a subset of these cases. For example, the result of merging one term specifying that all cases in suite S are to be executed, with another term specifying only test case X and Y in S, is a test of all cases in S. However, if a term specifying test case X and Y in S is merged with a term specifying case Z in S, the result is a test of X, Y, and Z in S. To disable this behavior, that is, to instead perform each test sequentially in a "script-like" manner, set term `merge_tests` to `false` in the test specification.

A test term can also specify one or more test suites, groups, or test cases to be skipped. Skipped suites, groups, and cases are not executed and show up in the HTML log files as `SKIPPED`.

Using Multiple Test Specification Files

When multiple test specification files are specified at startup (either with `ct_run -spec file1 file2 ...` or `ct:run_test([{spec, [File1,File2,...]}])`, `Common Test` either executes one test run per specification file, or joins the files and performs all tests within one single test run. The first behavior is the default one. The latter requires that start flag/option `join_specs` is provided, for example, `run_test -spec ./my_tests1.ts ./my_tests2.ts -join_specs`.

Joining a number of specifications, or running them separately, can also be accomplished with (and can be combined with) test specification file inclusion.

Test Specification File Inclusion

With the term `specs`, a test specification can include other specifications. An included specification can either be joined with the source specification or used to produce a separate test run (as with start flag/option `join_specs` above).

Example:

```
%% In specification file "a.spec"
{specs, join, ["b.spec", "c.spec"]}.
{specs, separate, ["d.spec", "e.spec"]}.
%% Config and test terms follow
...
```

In this example, the test terms defined in files "b.spec" and "c.spec" are joined with the terms in source specification "a.spec" (if any). The inclusion of specifications "d.spec" and "e.spec" results in two separate, and independent, test runs (one for each included specification).

Option `join` does not imply that the test terms are merged, only that all tests are executed in one single test run.

Joined specifications share common configuration settings, such as the list of `config` files or `include` directories. For configurations that cannot be combined, such as settings for `logdir` or `verbosity`, it is up to the user to ensure there are no clashes when the test specifications are joined. Specifications included with option `separate` do not share configuration settings with the source specification. This is useful, for example, if there are clashing configuration settings in included specifications, making it them impossible to join.

If `{merge_tests,true}` is set in the source specification (which is the default setting), terms in joined specifications are merged with terms in the source specification (according to the description of `merge_tests` earlier).

Notice that it is always the `merge_tests` setting in the source specification that is used when joined with other specifications. Say, for example, that a source specification A, with tests TA1 and TA2, has `{merge_tests,false}` set, and that it includes another specification, B, with tests TB1 and TB2, that has `{merge_tests,true}` set. The result is that the test series `TA1,TA2,merge(TB1,TB2)` is executed. The opposite `merge_tests` settings would result in the test series `merge(merge(TA1,TA2),TB1,TB2)`.

The term `specs` can be used to nest specifications, that is, have one specification include other specifications, which in turn include others, and so no

Test Case Groups

When a test case group is specified, the resulting test executes function `init_per_group`, followed by all test cases and subgroups (including their configuration functions), and finally function `end_per_group`. Also, if particular test cases in a group are specified, `init_per_group` and `end_per_group`, for the group in question, are called. If a group defined (in `Suite:group/0`) as a subgroup of another group, is specified (or if particular test cases of

a subgroup are), `Common Test` calls the configuration functions for the top-level groups and for the subgroup in question (making it possible to pass configuration data all the way from `init_per_suite` down to the test cases in the subgroup).

The test specification uses the same mechanism for specifying test case groups through names and paths, as explained in section *Test Case Group Execution*, with the addition of element `GroupSpec`.

Element `GroupSpec` makes it possible to specify group execution properties that overrides those in the group definition (that is, in `groups/0`). Execution properties for subgroups might be overridden as well. This feature makes it possible to change properties of groups at the time of execution, without having to edit the test suite. The same feature is available for `group` elements in the `Suite:all/0` list. For details and examples, see section *Test Case Groups*.

Test Specification Syntax

Test specifications can be used to run tests both in a single test host environment and in a distributed `Common Test` environment (Large Scale Testing). The node parameters in term `init` are only relevant in the latter (see section *Test Specifications* in Large Scale Testing). For details about the various terms, see the corresponding sections in the User's Guide, for example, the following:

- The *`ct_run` program* for an overview of available start flags (as most flags have a corresponding configuration term)
- *Logging* (for terms `verbosity`, `stylesheet`, `basic_html` and `esc_chars`)
- *External Configuration Data* (for terms `config` and `userconfig`)
- *Event Handling* (for the `event_handler` term)
- *Common Test Hooks* (for term `ct_hooks`)

Configuration terms:

```
{merge_tests, Bool}.

{define, Constant, Value}.

{specs, InclSpecsOption, TestSpecs}.

{node, NodeAlias, Node}.

{init, InitOptions}.
{init, [NodeAlias], InitOptions}.

{label, Label}.
{label, NodeRefs, Label}.

{verbosity, VerbosityLevels}.
{verbosity, NodeRefs, VerbosityLevels}.

{stylesheet, CSSFile}.
{stylesheet, NodeRefs, CSSFile}.

{silent_connections, ConnTypes}.
{silent_connections, NodeRefs, ConnTypes}.

{multiply_timetraps, N}.
{multiply_timetraps, NodeRefs, N}.

{scale_timetraps, Bool}.
{scale_timetraps, NodeRefs, Bool}.

{cover, CoverSpecFile}.
{cover, NodeRefs, CoverSpecFile}.

{cover_stop, Bool}.
{cover_stop, NodeRefs, Bool}.

{include, IncludeDirs}.
{include, NodeRefs, IncludeDirs}.

{auto_compile, Bool},
{auto_compile, NodeRefs, Bool},

{abort_if_missing_suites, Bool},
{abort_if_missing_suites, NodeRefs, Bool},

{config, ConfigFiles}.
{config, ConfigDir, ConfigBaseNames}.
{config, NodeRefs, ConfigFiles}.
{config, NodeRefs, ConfigDir, ConfigBaseNames}.

{userconfig, {CallbackModule, ConfigStrings}}.
{userconfig, NodeRefs, {CallbackModule, ConfigStrings}}.

{logdir, LogDir}.
{logdir, NodeRefs, LogDir}.

{logopts, LogOpts}.
{logopts, NodeRefs, LogOpts}.

{create_priv_dir, PrivDirOption}.
{create_priv_dir, NodeRefs, PrivDirOption}.

{event_handler, EventHandlers}.
{event_handler, NodeRefs, EventHandlers}.
{event_handler, EventHandlers, InitArgs}.
```

```
{event_handler, NodeRefs, EventHandlers, InitArgs}.

{ct_hooks, CTHModules}.
{ct_hooks, NodeRefs, CTHModules}.

{enable_builtin_hooks, Bool}.

{basic_html, Bool}.
{basic_html, NodeRefs, Bool}.

{esc_chars, Bool}.
{esc_chars, NodeRefs, Bool}.

{release_shell, Bool}.
```

Test terms:

```


{suites, Dir, Suites}.
{suites, NodeRefs, Dir, Suites}.

{groups, Dir, Suite, Groups}.
{groups, NodeRefs, Dir, Suite, Groups}.

{groups, Dir, Suite, Groups, {cases,Cases}}.
{groups, NodeRefs, Dir, Suite, Groups, {cases,Cases}}.

{cases, Dir, Suite, Cases}.
{cases, NodeRefs, Dir, Suite, Cases}.

{skip_suites, Dir, Suites, Comment}.
{skip_suites, NodeRefs, Dir, Suites, Comment}.

{skip_groups, Dir, Suite, GroupNames, Comment}.
{skip_groups, NodeRefs, Dir, Suite, GroupNames, Comment}.

{skip_cases, Dir, Suite, Cases, Comment}.
{skip_cases, NodeRefs, Dir, Suite, Cases, Comment}.
```

Types:

```
Bool            = true | false
Constant        = atom()
Value           = term()
InclSpecsOption = join | separate
TestSpecs       = string() | [string()]
NodeAlias       = atom()
Node            = node()
NodeRef         = NodeAlias | Node | master
NodeRefs        = all_nodes | [NodeRef] | NodeRef
InitOptions     = term()
Label           = atom() | string()
VerbosityLevels = integer() | [{Category,integer()}]
Category        = atom()
CSSFile         = string()
ConnTypes       = all | [atom()]
N               = integer()
CoverSpecFile   = string()
IncludeDirs     = string() | [string()]
ConfigFiles     = string() | [string()]
ConfigDir       = string()
ConfigBaseNames = string() | [string()]
CallbackModule  = atom()
ConfigStrings   = string() | [string()]
LogDir          = string()
LogOpts         = [term()]
PrivDirOption   = auto_per_run | auto_per_tc | manual_per_tc
EventHandlers   = atom() | [atom()]
InitArgs        = [term()]
CTHModules      = [CTHModule |
     {CTHModule, CTHInitArgs} |
     {CTHModule, CTHInitArgs, CTHPriority}]
CTHModule       = atom()
CTHInitArgs     = term()
Dir             = string()
Suites          = atom() | [atom()] | all
Suite           = atom()
Groups          = GroupPath | [GroupPath] | GroupSpec | [GroupSpec] | all
GroupPath       = [GroupName]
GroupSpec       = GroupName | {GroupName,Properties} | {GroupName,Properties,GroupSpec}
GroupName       = atom()
GroupNames      = GroupName | [GroupName]
Cases           = atom() | [atom()] | all
Comment         = string() | ""
```

The difference between the `config` terms above is that with `ConfigDir`, `ConfigBaseNames` is a list of base names, that is, without directory paths. `ConfigFiles` must be full names, including paths. For example, the following two terms have the same meaning:

```
{config, ["/home/testuser/tests/config/nodeA.cfg",
          "/home/testuser/tests/config/nodeB.cfg"]}.

{config, "/home/testuser/tests/config", ["nodeA.cfg","nodeB.cfg"]}.
```

Note:

Any relative paths, specified in the test specification, are relative to the directory containing the test specification file if `ct_run -spec TestSpecFile ...` or `ct:run:test([{spec,TestSpecFile},...])` executes the test.

The path is relative to the top-level log directory if `ct:run:testspec(TestSpec)` executes the test.

Constants

The term `define` introduces a constant that is used to replace the name `Constant` with `Value`, wherever it is found in the test specification. This replacement occurs during an initial iteration through the test specification. Constants can be used anywhere in the test specification, for example, in any lists and tuples, and even in strings and inside the value part of other constant definitions. A constant can also be part of a node name, but that is the only place where a constant can be part of an atom.

Note:

For the sake of readability, the name of the constant must always begin with an uppercase letter, or a $, ?, or _. This means that it must always be single quoted (as the constant name is an atom, not text).

The main benefit of constants is that they can be used to reduce the size (and avoid repetition) of long strings, such as file paths.

Examples:

```
%% 1a. no constant
{config, "/home/testuser/tests/config", ["nodeA.cfg","nodeB.cfg"]}.
{suites, "/home/testuser/tests/suites", all}.

%% 1b. with constant
{define, 'TESTDIR', "/home/testuser/tests"}.
{config, "'TESTDIR'/config", ["nodeA.cfg","nodeB.cfg"]}.
{suites, "'TESTDIR'/suites", all}.

%% 2a. no constants
{config, [testnode@host1, testnode@host2], "../config", ["nodeA.cfg","nodeB.cfg"]}.
{suites, [testnode@host1, testnode@host2], "../suites", [x_SUITE, y_SUITE]}.

%% 2b. with constants
{define, 'NODE', testnode}.
{define, 'NODES', ['NODE'@host1, 'NODE'@host2]}.
{config, 'NODES', "../config", ["nodeA.cfg","nodeB.cfg"]}.
{suites, 'NODES', "../suites", [x_SUITE, y_SUITE]}.
```

Constants make the test specification term `alias`, in previous versions of `Common Test`, redundant. This term is deprecated but remains supported in upcoming `Common Test` releases. Replacing `alias` terms with `define` is strongly recommended though. An example of such replacement follows:

```
%% using the old alias term
{config, "/home/testuser/tests/config/nodeA.cfg"}.
{alias, suite_dir, "/home/testuser/tests/suites"}.
{groups, suite_dir, x_SUITE, group1}.

%% replacing with constants
{define, 'TestDir', "/home/testuser/tests"}.
{define, 'CfgDir', "'TestDir'/config"}.
{define, 'SuiteDir', "'TestDir'/suites"}.
{config, 'CfgDir', "nodeA.cfg"}.
{groups, 'SuiteDir', x_SUITE, group1}.
```

Constants can well replace term `node` also, but this still has a declarative value, mainly when used in combination with `NodeRefs == all_nodes` (see *Types*).

Example

Here follows a simple test specification example:

```
{define, 'Top', "/home/test"}.
{define, 'T1', "'Top'/t1"}.
{define, 'T2', "'Top'/t2"}.
{define, 'T3', "'Top'/t3"}.
{define, 'CfgFile', "config.cfg"}.

{logdir, "'Top'/logs"}.

{config, ["'T1'/'CfgFile'", "'T2'/'CfgFile'", "'T3'/'CfgFile'"]}.

{suites, 'T1', all}.
{skip_suites, 'T1', [t1B_SUITE,t1D_SUITE], "Not implemented"}.
{skip_cases, 'T1', t1A_SUITE, [test3,test4], "Irrelevant"}.
{skip_cases, 'T1', t1C_SUITE, [test1], "Ignore"}.

{suites, 'T2', [t2B_SUITE,t2C_SUITE]}.
{cases, 'T2', t2A_SUITE, [test4,test1,test7]}.

{skip_suites, 'T3', all, "Not implemented"}.
```

The example specifies the following:

- The specified `logdir` directory is used for storing the HTML log files (in subdirectories tagged with node name, date, and time).
- The variables in the specified test system configuration files are imported for the test.
- The first test to run includes all suites for system `t1`. Suites `t1B` and `t1D` are excluded from the test. Test cases `test3` and `test4` in `t1A` and `test1` case in `t1C` are also excluded from the test.
- The second test to run is for system `t2`. The included suites are `t2B` and `t2C`. Test cases `test4`, `test1`, and `test7` in suite `t2A` are also included. The test cases are executed in the specified order.
- The last test to run is for system `t3`. Here, all suites are skipped and this is explicitly noted in the log files.

The init Term

With term `init` it is possible to specify initialization options for nodes defined in the test specification. There are options to start the node and to evaluate any function on the node. For details, see section *Automatic Startup of Test Target Nodes* in section Using Common Test for Large Scale Testing.

User-Specific Terms

The user can provide a test specification including (for `Common Test`) unrecognizable terms. If this is desired, use flag `-allow_user_terms` when starting tests with `ct_run`. This forces `Common Test` to ignore unrecognizable terms. In this mode, `Common Test` is not able to check the specification for errors as efficiently as if the scanner runs in default mode. If *`ct:run_test/1`* is used for starting the tests, the relaxed scanner mode is enabled by tuple `{allow_user_terms,true}`.

Reading Test Specification Terms

Terms in the current test specification (that is, the specification that has been used to configure and run the current test) can be looked up. The function *`get_testspec_terms()`* returns a list of all test specification terms (both configuration terms and test terms), and `get_testspec_terms(Tags)` returns the term (or a list of terms) matching the tag (or tags) in `Tags`.

For example, in the test specification:

```
...
{label, my_server_smoke_test}.
{config, "../../my_server_setup.cfg"}.
{config, "../../my_server_interface.cfg"}.
...
```

And in, for example, a test suite or a `Common Test Hook` function:

```
...
[{label,[{_Node,TestType}]}, {config,CfgFiles}] =
    ct:get_testspec_terms([label,config]),

[verify_my_server_cfg(TestType, CfgFile) || {Node,CfgFile} <- CfgFiles,
         Node == node()];
...
```

1.8.9 Log Files

As the execution of the test suites proceed, events are logged in the following four different ways:

- Text to the operator console.
- Suite-related information is sent to the major log file.
- Case-related information is sent to the minor log file.
- The HTML overview log file is updated with test results.
- A link to all runs executed from a certain directory is written in the log named `all_runs.html` and direct links to all tests (the latest results) are written to the top-level `index.html`.

Typically the operator, possibly running hundreds or thousands of test cases, does not want to fill the console with details about, or printouts from, specific test cases. By default, the operator only sees the following:

- A confirmation that the test has started and information about how many test cases are executed in total.
- A small note about each failed test case.
- A summary of all the run test cases.
- A confirmation when the test run is complete.
- Some special information, such as error reports, progress reports, and printouts written with `erlang:display/1`, or `io:format/3` specifically addressed to a receiver other than `standard_io` (for example, the default group leader process `user`).

To dig deeper into the general results, or the result of a specific test case, the operator can do so by following the links in the HTML presentation and read the major or minor log files. The "all_runs.html" page is a good starting point. It is located in `logdir` and contains a link to each test run, including a quick overview (with date and time, node name, number of tests, test names, and test result totals).

An "index.html" page is written for each test run (that is, stored in the `ct_run` directory tagged with node name, date, and time). This file provides an overview of all individual tests performed in the same test run. The test names follow the following convention:

- `TopLevelDir.TestDir` (all suites in `TestDir` executed)
- `TopLevelDir.TestDir:suites` (specific suites executed)
- `TopLevelDir.TestDir.Suite` (all cases in `Suite` executed)
- `TopLevelDir.TestDir.Suite:cases` (specific test cases executed)
- `TopLevelDir.TestDir.Suite.Case` (only `Case` executed)

The "test run index" page includes a link to the `Common Test` Framework Log file in which information about imported configuration data and general test progress is written. This log file is useful to get snapshot information about the test run during execution. It can also be helpful when analyzing test results or debugging test suites.

The "test run index" page indicates if a test has missing suites (that is, suites that `Common Test` failed to compile). Names of the missing suites can be found in the `Common Test` Framework Log file.

The major log file shows a detailed report of the test run. It includes test suite and test case names, execution time, the exact reason for failures, and so on. The information is available in both a file with textual and with HTML representation. The HTML file shows a summary that gives a good overview of the test run. It also has links to each individual test case log file for quick viewing with an HTML browser.

The minor log files contain full details of every single test case, each in a separate file. This way, it is straightforward to compare the latest results to that of previous test runs, even if the set of test cases changes. If application SASL is running, its logs are also printed to the current minor log file by the *cth_log_redirect built-in hook*.

The full name of the minor log file (that is, the name of the file including the absolute directory path) can be read during execution of the test case. It comes as value in tuple `{tc_logfile,LogFileName}` in the `Config` list (which means it can also be read by a pre- or post `Common Test Hook` function). Also, at the start of a test case, this data is sent with an event to any installed event handler. For details, see section *Event Handling*.

The log files are written continuously during a test run and links are always created initially when a test starts. Thevtest progress can therefore be followed simply by refreshing pages in the HTML browser. Statistics totals are not presented until a test is complete however.

Log Options

With start flag `logopts` options that modify some aspects of the logging behavior can be specified. The following options are available:

`no_src`

The HTML version of the test suite source code is not generated during the test run (and is consequently not available in the log file system).

`no_nl`

`Common Test` does not add a newline character `(\n)` to the end of an output string that it receives from a call to, for example, `io:format/2`, and which it prints to the test case log.

For example, if a test is started with:

```
$ ct_run -suite my_SUITE -logopts no_nl
```

then printouts during the test made by successive calls to `io:format("x")`, appears in the test case log as:

```
xxx
```

instead of each `x` printed on a new line, which is the default behavior.

Sorting HTML Table Columns

By clicking the name in the column header of any table (for example, "Ok", "Case", "Time", and so on), the table rows are sorted in whatever order makes sense for the type of value (for example, numerical for "Ok" or "Time", and alphabetical for "Case"). The sorting is performed through JavaScript code, automatically inserted into the HTML log files. `Common Test` uses the **jQuery** library and the **tablesorter** plugin, with customized sorting functions, for this implementation.

The Unexpected I/O Log

The test suites overview page includes a link to the Unexpected I/O Log. In this log, `Common Test` saves printouts made with *`ct:log/1,2,3,4,5`* and *`ct:pal/1,2,3,4,5`*, as well as captured system error- and progress reports, which cannot be associated with particular test cases and therefore cannot be written to individual test case log files. This occurs, for example, if a log printout is made from an external process (not a test case process), **or** if an error- or progress report comes in, during a short interval while `Common Test` is not executing a test case or configuration function, **or** while `Common Test` is currently executing a parallel test case group.

The Pre- and Post Test I/O Log

The `Common Test` Framework Log page includes links to the Pre- and Post Test I/O Log. In this log, `Common Test` saves printouts made with `ct:log/1,2,3,4,5` and `ct:pal/1,2,3,4,5`, as well as captured system error- and progress reports, which take place before, and after, the test run. Examples of this are printouts from a CT hook init- or terminate function, or progress reports generated when an OTP application is started from a CT hook init function. Another example is an error report generated because of a failure when an external application is stopped from a CT hook terminate function. All information in these examples ends up in the Pre- and Post Test I/O Log. For more information on how to synchronize test runs with external user applications, see section *Synchronizing* in section Common Test Hooks.

> **Note:**
>
> Logging to file with `ct:log/1,2,3,4,5` or `ct:pal/1,2,3,4,5` only works when `Common Test` is running. Printouts with `ct:pal/1,2,3,4,5` are however always displayed on screen.

Delete Old Logs

`Common Test` can automatically delete old log. This is specified with the `keep_logs` option. The default value for this option is `all`, which means that no logs are deleted. If the value is set to an integer, `N`, `Common Test` deletes all `ct_run.<timestamp>` directories, except the `N` newest.

1.8.10 HTML Style Sheets

`Common Test` uses an HTML Style Sheet (CSS file) to control the look of the HTML log files generated during test runs. If the log files are not displayed correctly in the browser of your choice, or you prefer a more primitive ("pre `Common Test` v1.6") look of the logs, use the start flag/option:

```
basic_html
```

This disables the use of style sheets and JavaScripts (see *Sorting HTML Table Columns*).

`Common Test` includes an **optional** feature to allow user HTML style sheets for customizing printouts. The functions in `ct` that print to a test case HTML log file (`log/3,4,5` and `pal/3,4,5`) accept `Category` as first argument. With this argument a category can be specified that can be mapped to a named `div` selector in a CSS rule-set. This is

useful, especially for coloring text differently depending on the type of (or reason for) the printout. Say you want one particular background color for test system configuration information, a different one for test system state information, and finally one for errors detected by the test case functions. The corresponding style sheet can look as follows:

```
div.sys_config  { background:blue }
div.sys_state   { background:yellow }
div.error       { background:red }
```

Common Test prints the text from `ct:log/3,4,5` or `ct:pal/3,4,5` inside a `pre` element nested under the named `div` element. Since the `pre` selector has a predefined CSS rule (in file `ct_default.css`) for the attributes `color`, `font-family` and `font-size`, if a user wants to change any of the predefined attribute settings, a new rule for `pre` must be added to the user stylesheet. Example:

```
div.error pre { color:white }
```

Here, white text is used instead of the default black for `div.error` printouts (and no other attribute settings for `pre` are affected).

To install the CSS file (`Common Test` inlines the definition in the HTML code), the file name can be provided when executing `ct_run`.

Example:

```
$ ct_run -dir $TEST/prog -stylesheet $TEST/styles/test_categories.css
```

Categories in a CSS file installed with flag `-stylesheet` are on a global test level in the sense that they can be used in any suite that is part of the test run.

Style sheets can also be installed on a per suite and per test case basis.

Example:

```
-module(my_SUITE).
...
suite() -> [..., {stylesheet,"suite_categories.css"}, ...].
...
my_testcase(_) ->
    ...
    ct:log(sys_config, "Test node version: ~p", [VersionInfo]),
    ...
    ct:log(sys_state, "Connections: ~p", [ConnectionInfo]),
    ...
    ct:pal(error, "Error ~p detected! Info: ~p", [SomeFault,ErrorInfo]),
    ct:fail(SomeFault).
```

If the style sheet is installed as in this example, the categories are private to the suite in question. They can be used by all test cases in the suite, but cannot be used by other suites. A suite private style sheet, if specified, is used in favor of a global style sheet (one specified with flag `-stylesheet`). A stylesheet tuple (as returned by `suite/0` above) can also be returned from a test case information function. In this case the categories specified in the style sheet can only be used in that particular test case. A test case private style sheet is used in favor of a suite or global level style sheet.

In a tuple `{stylesheet,CSSFile}`, if `CSSFile` is specified with a path, for example, `"$TEST/styles/categories.css"`, this full name is used to locate the file. However, if only the file name is specified, for example, `categories.css`, the CSS file is assumed to be located in the data directory, `data_dir`, of the suite. The latter use is recommended, as it is portable compared to hard coding path names in the suite.

Argument `Category` in the previous example can have the value (atom) `sys_config` (blue background), `sys_state` (yellow background), or `error` (white text on red background).

1.8.11 Repeating Tests

You can order `Common Test` to repeat the tests you specify. You can choose to repeat tests a number of times, repeat tests for a specific period of time, or repeat tests until a particular stop time is reached. If repetition is controlled by time, an action for `Common Test` to take upon time-out can be specified. Either `Common Test` performs all tests in the current run before stopping, or it stops when the current test job is finished. Repetition can be activated by `ct_run` start flags, or tuples in the `ct:run:test/1` option list argument. The flags (options in parentheses) are the following:

- `-repeat N ({repeat,N})`, where `N` is a positive integer
- `-duration DurTime ({duration,DurTime})`, where `DurTime` is the duration
- `-until StopTime ({until,StopTime})`, where `StopTime` is finish time
- `-force_stop ({force_stop,true})`
- `-force_stop skip_rest ({force_stop,skip_rest})`

`DurTime`

The duration time is specified as `HHMMSS`, for example, `-duration 012030` or `{duration,"012030"}`, which means that the tests are executed and (if time allows) repeated until time-out occurs after 1 hour, 20 minutes, and 30 seconds.

`StopTime`

The finish time can be specified as `HHMMSS` and is then interpreted as a time today (or possibly tomorrow), but can also be specified as `YYMoMoDDHHMMSS`, for example, `-until 071001120000` or `{until,"071001120000"}`. This means that the tests are executed and (if time allows) repeated, until 12 o'clock on the 1st of October 2007.

When time-out occurs, `Common Test` never aborts the ongoing test case, as this can leave the SUT in an undefined, and possibly bad, state. Instead `Common Test`, by default, finishes the current test run before stopping. If flag `force_stop` is specified, `Common Test` stops when the current test job is finished. If flag `force_stop` is specified with `skip_rest`, `Common Test` only completes the current test case and skips the remaining tests in the test job.

Note:

As `Common Test` always finishes at least the current test case, the time specified with `duration` or `until` is never definitive.

Log files from every repeated test run is saved in normal `Common Test` fashion (described earlier).

`Common Test` might later support an optional feature to only store the last (and possibly the first) set of logs of repeated test runs, but for now the user must be careful not to run out of disk space if tests are repeated during long periods of time.

For each test run that is part of a repeated session, information about the particular test run is printed in the `Common Test` Framework Log. The information includes the repetition number, remaining time, and so on.

Example 1:

```
$ ct_run -dir $TEST_ROOT/to1 $TEST_ROOT/to2 -duration 001000 -force_stop
```

Here, the suites in test directory `to1`, followed by the suites in `to2`, are executed in one test run. A time-out event occurs after 10 minutes. As long as there is time left, `Common Test` repeats the test run (that is, starting over with

test `to1`). After time-out, `Common Test` stops when the current job is finished (because of flag `force_stop`). As a result, the specified test run can be aborted after test `to1` and before test `to2`.

Example 2:

```
$ ct_run -dir $TEST_ROOT/to1 $TEST_ROOT/to2 -duration 001000 -forces_stop skip_rest
```

Here, the same tests as in Example 1 are run, but with flag `force_stop` set to `skip_rest`. If time-out occurs while executing tests in directory `to1`, the remaining test cases in `to1` are skipped and the test is aborted without running the tests in `to2` another time. If time-out occurs while executing tests in directory `to2`, the remaining test cases in `to2` are skipped and the test is aborted.

Example 3:

```
$ date
Fri Sep 28 15:00:00 MEST 2007

$ ct_run -dir $TEST_ROOT/to1 $TEST_ROOT/to2 -until 160000
```

Here, the same test run as in the previous examples are executed (and possibly repeated). However, when the time-out occurs, after 1 hour, `Common Test` finishes the entire test run before stopping (that is, both `to1` and `to2` are always executed in the same test run).

Example 4:

```
$ ct_run -dir $TEST_ROOT/to1 $TEST_ROOT/to2 -repeat 5
```

Here, the test run, including both the `to1` and the `to2` test, is repeated five times.

Note:

Do not confuse this feature with the `repeat` property of a test case group. The options described here are used to repeat execution of entire test runs, while the `repeat` property of a test case group makes it possible to repeat execution of sets of test cases within a suite. For more information about the latter, see section *Test Case Groups* in section Writing Test Suites.

1.8.12 Silent Connections

The protocol handling processes in `Common Test`, implemented by `ct_telnet`, `ct_ssh`, `ct_ftp`, and so on, do verbose printing to the test case logs. This can be switched off with flag `-silent_connections`:

```
ct_run -silent_connections [conn_types]
```

Here, `conn_types` specifies SSH, Telnet, FTP, RPC, and/or SNMP.

Example 1:

```
ct_run ... -silent_connections ssh telnet
```

This switches off logging for SSH and Telnet connections.

Example 2:

```
ct_run ... -silent_connections
```

This switches off logging for all connection types.

Fatal communication error and reconnection attempts are always printed, even if logging has been suppressed for the connection type in question. However, operations such as sending and receiving data are performed silently.

`silent_connections` can also be specified in a test suite. This is accomplished by returning a tuple, `{silent_connections,ConnTypes}`, in the `suite/0` or test case information list. If `ConnTypes` is a list of atoms (SSH, Telnet, FTP, RPC and/or SNMP), output for any corresponding connections are suppressed. Full logging is by default enabled for any connection of type not specified in `ConnTypes`. Hence, if `ConnTypes` is the empty list, logging is enabled for all connections.

Example 3:

```
-module(my_SUITE).

suite() -> [..., {silent_connections,[telnet,ssh]}, ...].

...

my_testcase1() ->
    [{silent_connections,[ssh]}].

my_testcase1(_) ->
    ...

my_testcase2(_) ->
    ...
```

In this example, `suite/0` tells `Common Test` to suppress printouts from Telnet and SSH connections. This is valid for all test cases. However, `my_testcase1/0` specifies that for this test case, only SSH is to be silent. The result is that `my_testcase1` gets Telnet information (if any) printed in the log, but not SSH information. `my_testcase2` gets no information from either connection printed.

`silent_connections` can also be specified with a term in a test specification (see section *Test Specifications* in section Running Tests and Analyzing Results). Connections provided with start flag/option `silent_connections` are merged with any connections listed in the test specification.

Start flag/option `silent_connections` and the test specification term override any settings made by the information functions inside the test suite.

Note:

In the current `Common Test` version, the `silent_connections` feature only works for Telnet and SSH connections. Support for other connection types can be added in future `Common Test` versions.

1.9 External Configuration Data

1.9.1 General

To avoid hard-coding data values related to the test and/or System Under Test (SUT) in the test suites, the data can instead be specified through configuration files or strings that `Common Test` reads before the start of a test run. External configuration data makes it possible to change test properties without modifying the test suites using the data. Examples of configuration data follows:

- Addresses to the test plant or other instruments
- User login information

- Names of files needed by the test
- Names of programs to be executed during the test
- Any other variable needed by the test

1.9.2 Syntax

A configuration file can contain any number of elements of the type:

```
{CfgVarName,Value}.
```

where

```
CfgVarName = atom()
Value = term() | [{CfgVarName,Value}]
```

1.9.3 Requiring and Reading Configuration Data

In a test suite, one must **require** that a configuration variable (`CfgVarName` in the previous definition) exists before attempting to read the associated value in a test case or configuration function.

`require` is an assert statement, which can be part of the *Test Suite Information Function* or *Test Case Information Function*. If the required variable is unavailable, the test is skipped (unless a default value has been specified, see section *Test Case Information Function* for details). Also, function *`ct:require/1/2`* can be called from a test case to check if a specific variable is available. The return value from this function must be checked explicitly and appropriate action be taken depending on the result (for example, to skip the test case if the variable in question does not exist).

A `require` statement in the test suite information case or test case information-list is to look like `{require,CfgVarName}` or `{require,AliasName,CfgVarName}`. The arguments `AliasName` and `CfgVarName` are the same as the arguments to *`ct:require/1,2`*. `AliasName` becomes an alias for the configuration variable, and can be used as reference to the configuration data value. The configuration variable can be associated with any number of alias names, but each name must be unique within the same test suite. The two main uses for alias names follows:

- To identify connections (described later).
- To help adapt configuration data to a test suite (or test case) and improve readability.

To read the value of a configuration variable, use function *`get_config/1,2,3`*.

Example:

```
suite() ->
    [{require, domain, 'CONN_SPEC_DNS_SUFFIX'}].

...

testcase(Config) ->
    Domain = ct:get_config(domain),
    ...
```

1.9.4 Using Configuration Variables Defined in Multiple Files

If a configuration variable is defined in multiple files and you want to access all possible values, use function *`ct:get_config/3`* and specify `all` in the options list. The values are then returned in a list and the order of the elements corresponds to the order that the configuration files were specified at startup.

1.9.5 Encrypted Configuration Files

Configuration files containing sensitive data can be encrypted if they must be stored in open and shared directories.

To have `Common Test` encrypt a specified file using function `DES3` in application `Crypto`, call *`ct:encrypt_config_file/2,3`* The encrypted file can then be used as a regular configuration file in combination with other encrypted files or normal text files. However, the key for decrypting the configuration file must be provided when running the test. This can be done with flag/option `decrypt_key` or `decrypt_file`, or a key file in a predefined location.

`Common Test` also provides decryption functions, *`ct:decrypt_config_file/2,3`*, for recreating the original text files.

1.9.6 Opening Connections Using Configuration Data

Two different methods for opening a connection using the support functions in, for example, *`ct_ssh`*, *`ct_ftp`*, and *`ct_telnet`* follows:

- Using a configuration target name (an alias) as reference.
- Using the configuration variable as reference.

When a target name is used for referencing the configuration data (that specifies the connection to be opened), the same name can be used as connection identity in all subsequent calls related to the connection (also for closing it). Only one open connection per target name is possible. If you attempt to open a new connection using a name already associated with an open connection, `Common Test` returns the already existing handle so the previously opened connection is used. This feature makes it possible to call the function for opening a particular connection whenever useful. An action like this does not necessarily open any new connections unless it is required (which could be the case if, for example, the previous connection has been closed unexpectedly by the server). Using named connections also removes the need to pass handle references around in the suite for these connections.

When a configuration variable name is used as reference to the data specifying the connection, the handle returned as a result of opening the connection must be used in all subsequent calls (also for closing the connection). Repeated calls to the open function with the same variable name as reference results in multiple connections being opened. This can be useful, for example, if a test case needs to open multiple connections to the same server on the target node (using the same configuration data for each connection).

1.9.7 User-Specific Configuration Data Formats

The user can specify configuration data on a different format than key-value tuples in a text file, as described so far. The data can, for example, be read from any files, fetched from the web over HTTP, or requested from a user-specific process. To support this, `Common Test` provides a callback module plugin mechanism to handle configuration data.

Default Callback Modules for Handling Configuration Data

`Common Test` includes default callback modules for handling configuration data specified in standard configuration files (described earlier) and in XML files as follows:

- `ct_config_plain` - for reading configuration files with key-value tuples (standard format). This handler is used to parse configuration files if no user callback is specified.
- `ct_config_xml` - for reading configuration data from XML files.

Using XML Configuration Files

An example of an XML configuration file follows:

```
<config>
   <ftp_host>
       <ftp>"targethost"</ftp>
       <username>"tester"</username>
       <password>"letmein"</password>
   </ftp_host>
   <lm_directory>"/test/loadmodules"</lm_directory>
</config>
```

Once read, this file produces the same configuration variables as the following text file:

```
{ftp_host, [{ftp,"targethost"},
            {username,"tester"},
            {password,"letmein"}]}.

{lm_directory, "/test/loadmodules"}.
```

Implement a User-Specific Handler

The user-specific handler can be written to handle special configuration file formats. The parameter can be either file names or configuration strings (the empty list is valid).

The callback module implementing the handler is responsible for checking the correctness of configuration strings.

To validate the configuration strings, the callback module is to have function `Callback:check_parameter/1` exported.

The input argument is passed from `Common Test`, as defined in the test specification, or specified as an option to `ct_run` or `ct:run_test`.

The return value is to be any of the following values, indicating if the specified configuration parameter is valid:

- `{ok, {file, FileName}}` - the parameter is a file name and the file exists.
- `{ok, {config, ConfigString}}` - the parameter is a configuration string and it is correct.
- `{error, {nofile, FileName}}` - there is no file with the specified name in the current directory.
- `{error, {wrong_config, ConfigString}}` - the configuration string is wrong.

The function `Callback:read_config/1` is to be exported from the callback module to read configuration data, initially before the tests start, or as a result of data being reloaded during test execution. The input argument is the same as for function `check_parameter/1`.

The return value is to be either of the following:

- `{ok, Config}` - if the configuration variables are read successfully.
- `{error, {Error, ErrorDetails}}` - if the callback module fails to proceed with the specified configuration parameters.

`Config` is the proper Erlang key-value list, with possible key-value sublists as values, like the earlier configuration file example:

```
[{ftp_host, [{ftp, "targethost"}, {username, "tester"}, {password, "letmein"}]},
 {lm_directory, "/test/loadmodules"}]
```

1.9.8 Examples of Configuration Data Handling

A configuration file for using the FTP client to access files on a remote host can look as follows:

```
{ftp_host, [{ftp,"targethost"},
     {username,"tester"},
     {password,"letmein"}]}.

{lm_directory, "/test/loadmodules"}.
```

The XML version shown earlier can also be used, but it is to be explicitly specified that the `ct_config_xml` callback module is to be used by `Common Test`.

The following is an example of how to assert that the configuration data is available and can be used for an FTP session:

```
init_per_testcase(ftptest, Config) ->
    {ok,_} = ct_ftp:open(ftp),
    Config.

end_per_testcase(ftptest, _Config) ->
    ct_ftp:close(ftp).

ftptest() ->
    [{require,ftp,ftp_host},
     {require,lm_directory}].

ftptest(Config) ->
    Remote = filename:join(ct:get_config(lm_directory), "loadmodX"),
    Local = filename:join(?config(priv_dir,Config), "loadmodule"),
    ok = ct_ftp:recv(ftp, Remote, Local),
    ...
```

The following is an example of how the functions in the previous example can be rewritten if it is necessary to open multiple connections to the FTP server:

```
init_per_testcase(ftptest, Config) ->
    {ok,Handle1} = ct_ftp:open(ftp_host),
    {ok,Handle2} = ct_ftp:open(ftp_host),
    [{ftp_handles,[Handle1,Handle2]} | Config].

end_per_testcase(ftptest, Config) ->
    lists:foreach(fun(Handle) -> ct_ftp:close(Handle) end,
                  ?config(ftp_handles,Config)).

ftptest() ->
    [{require,ftp_host},
     {require,lm_directory}].

ftptest(Config) ->
    Remote = filename:join(ct:get_config(lm_directory), "loadmodX"),
    Local = filename:join(?config(priv_dir,Config), "loadmodule"),
    [Handle | MoreHandles] = ?config(ftp_handles,Config),
    ok = ct_ftp:recv(Handle, Remote, Local),
    ...
```

1.9.9 Example of User-Specific Configuration Handler

A simple configuration handling driver, asking an external server for configuration data, can be implemented as follows:

```
-module(config_driver).
-export([read_config/1, check_parameter/1]).

read_config(ServerName)->
    ServerModule = list_to_atom(ServerName),
    ServerModule:start(),
    ServerModule:get_config().

check_parameter(ServerName)->
    ServerModule = list_to_atom(ServerName),
    case code:is_loaded(ServerModule) of
        {file, _}->
            {ok, {config, ServerName}};
        false->
            case code:load_file(ServerModule) of
                {module, ServerModule}->
                    {ok, {config, ServerName}};
                {error, nofile}->
                    {error, {wrong_config, "File not found: " ++ ServerName ++ ".beam"}}
            end
    end.
```

The configuration string for this driver can be `config_server`, if the `config_server.erl` module that follows is compiled and exists in the code path during test execution:

```
-module(config_server).
-export([start/0, stop/0, init/1, get_config/0, loop/0]).

-define(REGISTERED_NAME, ct_test_config_server).

start()->
    case whereis(?REGISTERED_NAME) of
        undefined->
            spawn(?MODULE, init, [?REGISTERED_NAME]),
            wait();
        _Pid->
        ok
    end,
    ?REGISTERED_NAME.

init(Name)->
    register(Name, self()),
    loop().

get_config()->
    call(self(), get_config).

stop()->
    call(self(), stop).

call(Client, Request)->
    case whereis(?REGISTERED_NAME) of
        undefined->
            {error, {not_started, Request}};
        Pid->
            Pid ! {Client, Request},
            receive
                Reply->
                    {ok, Reply}
            after 4000->
                {error, {timeout, Request}}
            end
    end.

loop()->
    receive
        {Pid, stop}->
            Pid ! ok;
        {Pid, get_config}->
            {D,T} = erlang:localtime(),
            Pid !
                [{localtime, [{date, D}, {time, T}]},
                 {node, erlang:node()},
                 {now, erlang:now()},
                 {config_server_pid, self()},
                 {config_server_vsn, ?vsn}],
            ?MODULE:loop()
    end.

wait()->
    case whereis(?REGISTERED_NAME) of
        undefined->
            wait();
        _Pid->
            ok
    end.
```

Here, the handler also provides for dynamically reloading of configuration variables. If *ct:reload_config(localtime)* is called from the test case function, all variables loaded with config_driver:read_config/1 are updated with their latest values, and the new value for variable localtime is returned.

1.10 Code Coverage Analysis

1.10.1 General

Although Common Test was created primarily for black-box testing, nothing prevents it from working perfectly as a white-box testing tool as well. This is especially true when the application to test is written in Erlang. Then the test ports are easily realized with Erlang function calls.

When white-box testing an Erlang application, it is useful to be able to measure the code coverage of the test. Common Test provides simple access to the OTP Cover tool for this purpose. Common Test handles all necessary communication with the Cover tool (starting, compiling, analysing, and so on). The Common Test user only needs to specify the extent of the code coverage analysis.

1.10.2 Use

To specify the modules to be included in the code coverage test, provide a cover specification file. With this file you can point out specific modules or specify directories containing modules to be included in the analysis. You can also specify modules to be excluded from the analysis.

If you are testing a distributed Erlang application, it is likely that code you want included in the code coverage analysis gets executed on another Erlang node than the one Common Test is running on. If so, you must specify these other nodes in the cover specification file or add them dynamically to the code coverage set of nodes. For details on the latter, see module *ct_cover*.

In the cover specification file you can also specify your required level of the code coverage analysis; details or overview. In detailed mode, you get a coverage overview page, showing per module and total coverage percentages. You also get an HTML file printed for each module included in the analysis showing exactly what parts of the code have been executed during the test. In overview mode, only the code coverage overview page is printed.

You can choose to export and import code coverage data between tests. If you specify the name of an export file in the cover specification file, Common Test exports collected coverage data to this file at the end of the test. You can similarly specify previously exported data to be imported and included in the analysis for a test (multiple import files can be specified). This way, the total code coverage can be analyzed without necessarily running all tests at once.

To activate the code coverage support, specify the name of the cover specification file as you start Common Test. Do this by using flag -cover with *ct_run*, for example:

```
$ ct_run -dir $TESTOBJS/db -cover $TESTOBJS/db/config/db.coverspec
```

You can also pass the cover specification file name in a call to *ct:run_test/1*, by adding a {cover,CoverSpec} tuple to argument Opts.

You can also enable code coverage in your test specifications (see section *Test Specifications* in section Running Tests and Analyzing Results).

1.10.3 Stopping the Cover Tool When Tests Are Completed

By default, the Cover tool is automatically stopped when the tests are completed. This causes the original (non-cover compiled) modules to be loaded back into the test node. If a process at this point still runs old code of any of the modules that are cover compiled, meaning that it has not done any fully qualified function call after the cover compilation, the process is killed. To avoid this, set the value of option cover_stop to false. This means that the modules stay

cover compiled. Therefore, this is only recommended if the Erlang nodes under test are terminated after the test is completed, or if cover can be manually stopped.

The option can be set by using flag `-cover_stop` with `ct_run`, by adding `{cover_stop,true|false}` to argument `Opts` to *`ct:run_test/1`*, or by adding a `cover_stop` term in the test specification (see section *Test Specifications* in section Running Tests and Analyzing Results).

1.10.4 The Cover Specification File

The following terms are allowed in a cover specification file:

```
%% List of Nodes on which cover will be active during test.
%% Nodes = [atom()]
{nodes, Nodes}.

%% Files with previously exported cover data to include in analysis.
%% CoverDataFiles = [string()]
{import, CoverDataFiles}.

%% Cover data file to export from this session.
%% CoverDataFile = string()
{export, CoverDataFile}.

%% Cover analysis level.
%% Level = details | overview
{level, Level}.

%% Directories to include in cover.
%% Dirs = [string()]
{incl_dirs, Dirs}.

%% Directories, including subdirectories, to include.
{incl_dirs_r, Dirs}.

%% Specific modules to include in cover.
%% Mods = [atom()]
{incl_mods, Mods}.

%% Directories to exclude in cover.
{excl_dirs, Dirs}.

%% Directories, including subdirectories, to exclude.
{excl_dirs_r, Dirs}.

%% Specific modules to exclude in cover.
{excl_mods, Mods}.

%% Cross cover compilation
%% Tag = atom(), an identifier for a test run
%% Mod = [atom()], modules to compile for accumulated analysis
{cross,[{Tag,Mods}]}.
```

The terms `incl_dirs_r` and `excl_dirs_r` tell `Common Test` to search the specified directories recursively and include or exclude any module found during the search. The terms `incl_dirs` and `excl_dirs` result in a non-recursive search for modules (that is, only modules found in the specified directories are included or excluded).

Note:

Directories containing Erlang modules to be included in a code coverage test must exist in the code server path. Otherwise, the Cover tool fails to recompile the modules. It is not sufficient to specify these directories in the cover specification file for `Common Test`.

1.10.5 Cross Cover Analysis

The cross cover mechanism allows cover analysis of modules across multiple tests. It is useful if some code, for example, a library module, is used by many different tests and the accumulated cover result is desirable.

This can also be achieved in a more customized way by using parameter `export` in the cover specification and analysing the result off line. However, the cross cover mechanism is a built-in solution that also provides logging.

The mechanism is easiest explained by an example:

Assume that there are two systems, `s1` and `s2`, that are tested in separate test runs. System `s1` contains a library module `m1` tested by test run `s1` and is included in the cover specification of `s1` as follows:

```
s1.cover:
  {incl_mods,[m1]}.
```

When analysing code coverage, the result for `m1` can be seen in the cover log in the `s1` test result.

Now, imagine that as `m1` is a library module, it is also often used by system `s2`. Test run `s2` does not specifically test `m1`, but it can still be interesting to see which parts of `m1` that are covered by the `s2` tests. To do this, `m1` can be included also in the cover specification of `s2` as follows:

```
s2.cover:
  {incl_mods,[m1]}.
```

This gives an entry for `m1` also in the cover log for test run `s2`. The problem is that this only reflects the coverage by `s2` tests, not the accumulated result over `s1` and `s2`. This is where the cross cover mechanism comes in handy.

If instead the cover specification for `s2` is like the following:

```
s2.cover:
  {cross,[{s1,[m1]}]}.
```

Then `m1` is cover compiled in test run `s2`, but not shown in the coverage log. Instead, if *`ct_cover:cross_cover_analyse/2`* is called after both `s1` and `s2` test runs are completed, the accumulated result for `m1` is available in the cross cover log for test run `s1`.

The call to the analyze function must be as follows:

```
ct_cover:cross_cover_analyse(Level, [{s1,S1LogDir},{s2,S2LogDir}]).
```

Here, `S1LogDir` and `S2LogDir` are the directories named `<TestName>.logs` for each test respectively.

Notice the tags `s1` and `s2`, which are used in the cover specification file and in the call to `ct_cover:cross_cover_analyse/2`. The purpose of these is only to map the modules specified in the cover specification to the log directory specified in the call to the analyze function. The tag name has no meaning beyond this.

1.10.6 Logging

To view the result of a code coverage test, click the button labeled "COVER LOG" in the top-level index page for the test run.

Before Erlang/OTP 17.1, if your test run consisted of multiple tests, cover would be started and stopped for each test within the test run. Separate logs would be available through the "Coverage log" link on the test suite result pages. These links are still available, but now they all point to the same page as the button on the top-level index page. The log contains the accumulated results for the complete test run. For details about this change, see the release notes.

The button takes you to the code coverage overview page. If you have successfully performed a detailed coverage analysis, links to each individual module coverage page are found here.

If cross cover analysis is performed, and there are accumulated coverage results for the current test, the link "Coverdata collected over all tests" takes you to these results.

1.11 Using Common Test for Large-Scale Testing

1.11.1 General

Large-scale automated testing requires running multiple independent test sessions in parallel. This is accomplished by running some `Common Test` nodes on one or more hosts, testing different target systems. Configuring, starting, and controlling the test nodes independently can be a cumbersome operation. To aid this kind of automated large-scale testing, `Common Test` offers a master test node component, `Common Test` Master, which handles central configuration and control in a system of distributed `Common Test` nodes.

The `Common Test` Master server runs on one dedicated Erlang node and uses distributed Erlang to communicate with any number of `Common Test` test nodes, each hosting a regular `Common Test` server. Test specifications are used as input to specify what to test on which test nodes, using what configuration.

The `Common Test` Master server writes progress information to HTML log files similarly to the regular `Common Test` server. The logs contain test statistics and links to the log files written by each independent `Common Test` server.

The `Common Test` Master API is exported by module *`ct_master`*.

1.11.2 Use

`Common Test` Master requires all test nodes to be on the same network and share a common file system. `Common Test` Master cannot start test nodes automatically. The nodes must be started in advance for `Common Test` Master to be able to start test sessions on them.

Tests are started by calling *`ct_master:run(TestSpecs)`* or *`ct_master:run(TestSpecs, InclNodes, ExclNodes)`*

`TestSpecs` is either the name of a test specification file (string) or a list of test specifications. If it is a list, the specifications are handled (and the corresponding tests executed) in sequence. An element in a `TestSpecs` list can also be list of test specifications. The specifications in such a list are merged into one combined specification before test execution.

Example:

```
ct_master:run(["ts1","ts2",["ts3","ts4"]])
```

Here, the tests specified by "ts1" run first, then the tests specified by "ts2", and finally the tests specified by both "ts3" and "ts4".

The `InclNodes` argument to `run/3` is a list of node names. Function `run/3` runs the tests in `TestSpecs` just like `run/1`, but also takes any test in `TestSpecs`, which is not explicitly tagged with a particular node name, and execute it on the nodes listed in `InclNodes`. By using `run/3` this way, any test specification can be used, with or without node information, in a large-scale test environment.

`ExclNodes` is a list of nodes to be excluded from the test. That is, tests that are specified in the test specification to run on a particular node are not performed if that node is listed in `ExclNodes` at runtime.

If `Common Test` Master fails initially to connect to any of the test nodes specified in a test specification or in the `InclNodes` list, the operator is prompted with the option to either start over again (after manually checking the status of the nodes in question), to run without the missing nodes, or to abort the operation.

When tests start, `Common Test` Master displays information to console about the involved nodes. `Common Test` Master also reports when tests finish, successfully or unsuccessfully. If connection is lost to a node, the test on that node is considered finished. `Common Test` Master does not attempt to re-establish contact with the failing node.

At any time, to get the current status of the test nodes, call function *`ct_master:progress()`*.

To stop one or more tests, use function *`ct_master:abort()`* (to stop all) or *`ct_master:abort(Nodes)`*.

For details about the `Common Test` Master API, see module *`ct_master`*.

1.11.3 Test Specifications

The test specifications used as input to `Common Test` Master are fully compatible with the specifications used as input to the regular `Common Test` server. The syntax is described in section *Test Specifications* in section Running Tests and Analyzing Results.

All test specification terms can have a `NodeRefs` element. This element specifies which node or nodes a configuration operation or a test is to be executed on. `NodeRefs` is defined as follows:

```
NodeRefs = all_nodes | [NodeRef] | NodeRef
```

```
NodeRef = NodeAlias | node() | master
```

A `NodeAlias` (`atom()`) is used in a test specification as a reference to a node name (so the node name only needs to be declared once, which also can be achieved using constants). The alias is declared with a `node` term as follows:

```
{node, NodeAlias, NodeName}
```

If `NodeRefs` has the value `all_nodes`, the operation or test is performed on all specified test nodes. (Declaring a term without a `NodeRefs` element has the same effect). If `NodeRefs` has the value `master`, the operation is only performed on the `Common Test` Master node (namely set the log directory or install an event handler).

Consider the example in section *Test Specifications* in section Running Tests and Analysing Results, now extended with node information and intended to be executed by `Common Test` Master:

```
{define, 'Top', "/home/test"}.
{define, 'T1', "'Top'/t1"}.
{define, 'T2', "'Top'/t2"}.
{define, 'T3', "'Top'/t3"}.
{define, 'CfgFile', "config.cfg"}.
{define, 'Node', ct_node}.

{node, node1, 'Node@host_x'}.
{node, node2, 'Node@host_y'}.

{logdir, master, "'Top'/master_logs"}.
{logdir, "'Top'/logs"}.

{config, node1, "'T1'/'CfgFile'"}.
{config, node2, "'T2'/'CfgFile'"}.
{config, "'T3'/'CfgFile'"}.

{suites, node1, 'T1', all}.
{skip_suites, node1, 'T1', [t1B_SUITE,t1D_SUITE], "Not implemented"}.
{skip_cases, node1, 'T1', t1A_SUITE, [test3,test4], "Irrelevant"}.
{skip_cases, node1, 'T1', t1C_SUITE, [test1], "Ignore"}.

{suites, node2, 'T2', [t2B_SUITE,t2C_SUITE]}.
{cases, node2, 'T2', t2A_SUITE, [test4,test1,test7]}.

{skip_suites, 'T3', all, "Not implemented"}.
```

This example specifies the same tests as the original example. But now if started with a call to `ct_master:run(TestSpecName)`, test `t1` is executed on node `ct_node@host_x` (`node1`), test `t2` on `ct_node@host_y` (`node2`) and test `t3` on both `node1` and `node2`. Configuration file `t1` is only read on `node1` and configuration file `t2` only on `node2`, while the configuration file `t3` is read on both `node1` and `node2`. Both test nodes write log files to the same directory. (However, the `Common Test` Master node uses a different log directory than the test nodes.)

If the test session is instead started with a call to `ct_master:run(TestSpecName, [ct_node@host_z], [ct_node@host_x])`, the result is that test `t1` does not run on `ct_node@host_x` (or any other node) while test `t3` runs on both `ct_node@host_y` and `ct_node@host_z`.

A nice feature is that a test specification that includes node information can still be used as input to the regular `Common Test` server (as described in section *Test Specifications*). The result is that any test specified to run on a node with the same name as the `Common Test` node in question (typically `ct@somehost` if started with the `ct_run` program), is performed. Tests without explicit node association are always performed too, of course.

1.11.4 Automatic Startup of Test Target Nodes

Initial actions can be started and performed automatically on test target nodes using test specification term `init`.

Two subterms are supported, `node_start` and `eval`.

Example:

```
{node, node1, node1@host1}.
{node, node2, node1@host2}.
{node, node3, node2@host2}.
{node, node4, node1@host3}.
{init, node1, [{node_start, [{callback_module, my_slave_callback}]}]}.
{init, [node2, node3], {node_start, [{username, "ct_user"}, {password, "ct_password"}]}}.
{init, node4, {eval, {module, function, []}}}.
```

This test specification declares that `node1@host1` is to be started using the user callback function `callback_module:my_slave_callback/0`, and nodes `node1@host2` and `node2@host2` are to be started with the default callback module `ct_slave`. The specified username and password are used to log on to remote host `host2`. Also, function `module:function/0` is evaluated on `node1@host3`, and the result of this call is printed to the log.

The default callback module *ct_slave*, has the following features:

- Starting Erlang target nodes on local or remote hosts (application `SSH` is used for communication).
- Ability to start an Erlang emulator with more flags (any flags supported by `erl` are supported).
- Supervision of a node being started using internal callback functions. Used to prevent hanging nodes. (Configurable.)
- Monitoring of the master node by the slaves. A slave node can be stopped if the master node terminates. (Configurable.)
- Execution of user functions after a slave node is started. Functions can be specified as a list of `{Module, Function, Arguments}` tuples.

Note:

An `eval` term for the node and `startup_functions` in the `node_start` options list can be specified. In this case, the node is started first, then the `startup_functions` are executed, and finally functions specified with `eval` are called.

1.12 Event Handling

1.12.1 General

The operator of a `Common Test` system can receive event notifications continuously during a test run. For example, `Common Test` reports when a test case starts and stops, the current count of successful, failed, and skipped cases, and so on. This information can be used for different purposes such as logging progress and results in another format than HTML, saving statistics to a database for report generation, and test system supervision.

`Common Test` has a framework for event handling based on the OTP event manager concept and `gen_event` behavior. When the `Common Test` server starts, it spawns an event manager. During test execution the manager gets a notification from the server when something of potential interest happens. Any event handler plugged into the event manager can match on events of interest, take action, or pass the information on. The event handlers are Erlang modules implemented by the `Common Test` user according to the `gen_event` behavior (for details, see module *`gen_event`* and section *`gen_event Behaviour`* in OTP Design Principles in the System Documentation).

A `Common Test` server always starts an event manager. The server also plugs in a default event handler, which only purpose is to relay notifications to a globally registered `Common Test` Master event manager (if a `Common Test` Master server is running in the system). The `Common Test` Master also spawns an event manager at startup. Event handlers plugged into this manager receives the events from all the test nodes, plus information from the `Common Test` Master server.

User-specific event handlers can be plugged into a `Common Test` event manager, either by telling `Common Test` to install them before the test run (described later), or by adding the handlers dynamically during the test run using *`gen_event:add_handler/3`* or *`gen_event:add_sup_handler/3`*. In the latter scenario, the reference of the `Common Test` event manager is required. To get it, call *`ct:get_event_mgr_ref/0`* or (on the `Common Test` Master node) *`ct_master:get_event_mgr_ref/0`*.

1.12.2 Use

Event handlers can be installed by an `event_handler` start flag (*`ct_run`*) or option *`ct:run_test/1`*, where the argument specifies the names of one or more event handler modules.

Example:

```
$ ct_run -suite test/my_SUITE -event_handler handlers/my_evh1 handlers/my_evh2
-pa $PWD/handlers
```

To pass start arguments to the event handler init function, use option `ct_run -event_handler_init` instead of `-event_handler`.

> **Note:**
>
> All event handler modules must have `gen_event` behavior. These modules must be precompiled and their locations must be added explicitly to the Erlang code server search path (as in the previous example).

An event_handler tuple in argument `Opts` has the following definition (see *`ct:run_test/1`*):

```
{event_handler,EventHandlers}

EventHandlers = EH | [EH]
EH = atom() | {atom(),InitArgs} | {[atom()],InitArgs}
InitArgs = [term()]
```

In the following example, two event handlers for the `my_SUITE` test are installed:

```
1> ct:run_test([{suite,"test/my_SUITE"},{event_handler,[my_evh1,{my_evh2,[node()]}]}]).
```

Event handler `my_evh1` is started with `[]` as argument to the init function. Event handler `my_evh2` is started with the name of the current node in the init argument list.

Event handlers can also be plugged in using one of the following *test specification* terms:

- `{event_handler, EventHandlers}`
- `{event_handler, EventHandlers, InitArgs}`
- `{event_handler, NodeRefs, EventHandlers}`
- `{event_handler, NodeRefs, EventHandlers, InitArgs}`

`EventHandlers` is a list of module names. Before a test session starts, the init function of each plugged in event handler is called (with the `InitArgs` list as argument or `[]` if no start arguments are specified).

To plug in a handler to the `Common Test` Master event manager, specify `master` as the node in `NodeRefs`.

To be able to match on events, the event handler module must include the header file `ct_event.hrl`. An event is a record with the following definition:

```
#event{name, node, data}
```

`name`

Label (type) of the event.

`node`

Name of the node that the event originated from (only relevant for `Common Test` Master event handlers).

`data`

Specific for the event.

General Events

The general events are as follows:

`#event{name = start_logging, data = LogDir}`

`LogDir = string()`, top-level log directory for the test run.

This event indicates that the logging process of `Common Test` has started successfully and is ready to receive I/O messages.

`#event{name = stop_logging, data = []}`

This event indicates that the logging process of `Common Test` was shut down at the end of the test run.

`#event{name = test_start, data = {StartTime,LogDir}}`

`StartTime = {date(),time()}`, test run start date and time.

`LogDir = string()`, top-level log directory for the test run.

This event indicates that `Common Test` has finished initial preparations and begins executing test cases.

`#event{name = test_done, data = EndTime}`

`EndTime = {date(),time()}`, date and time the test run finished.

This event indicates that the last test case has been executed and `Common Test` is shutting down.

`#event{name = start_info, data = {Tests,Suites,Cases}}`

`Tests = integer()`, number of tests.

`Suites = integer()`, total number of suites.

`Cases = integer() | unknown`, total number of test cases.

This event gives initial test run information that can be interpreted as: "This test run will execute `Tests` separate tests, in total containing `Cases` number of test cases, in `Suites` number of suites". However, if a test case group with a repeat property exists in any test, the total number of test cases cannot be calculated (unknown).

`#event{name = tc_start, data = {Suite,FuncOrGroup}}`

`Suite = atom()`, name of the test suite.

`FuncOrGroup = Func | {Conf,GroupName,GroupProperties}`

`Func = atom()`, name of test case or configuration function.

`Conf = init_per_group | end_per_group`, group configuration function.

`GroupName = atom()`, name of the group.

`GroupProperties = list()`, list of execution properties for the group.

This event informs about the start of a test case, or a group configuration function. The event is sent also for `init_per_suite` and `end_per_suite`, but not for `init_per_testcase` and `end_per_testcase`. If a group configuration function starts, the group name and execution properties are also specified.

`#event{name = tc_logfile, data = {{Suite,Func},LogFileName}}`

`Suite = atom()`, name of the test suite.

`Func = atom()`, name of test case or configuration function.

`LogFileName = string()`, full name of the test case log file.

This event is sent at the start of each test case (and configuration function except `init/end_per_testcase`) and carries information about the full name (that is, the file name including the absolute directory path) of the current test case log file.

`#event{name = tc_done, data = {Suite,FuncOrGroup,Result}}`

`Suite = atom()`, name of the suite.

`FuncOrGroup = Func | {Conf,GroupName,GroupProperties}`

`Func = atom()`, name of test case or configuration function.

`Conf = init_per_group | end_per_group`, group configuration function.

`GroupName = unknown | atom()`, name of the group (unknown if init- or end function times out).

`GroupProperties = list()`, list of execution properties for the group.

`Result = ok | {auto_skipped,SkipReason} | {skipped,SkipReason} | {failed,FailReason}`, the result.

`SkipReason = {require_failed,RequireInfo} | {require_failed_in_suite0,RequireInfo} | {failed,{Suite,init_per_testcase,FailInfo}} | UserTerm`, why the case was skipped.

`FailReason = {error,FailInfo} | {error,{RunTimeError,StackTrace}} | {timetrap_timeout,integer()} | {failed,{Suite,end_per_testcase,FailInfo}}`, reason for failure.

`RequireInfo = {not_available,atom() | tuple()}`, why require failed.

`FailInfo = {timetrap_timeout,integer()} | {RunTimeError,StackTrace} | UserTerm`, error details.

`RunTimeError = term()`, a runtime error, for example, `badmatch` or `undef`.

`StackTrace = list()`, list of function calls preceding a runtime error.

`UserTerm = term()`, any data specified by user, or `exit/1` information.

This event informs about the end of a test case or a configuration function (see event `tc_start` for details on element `FuncOrGroup`). With this event comes the final result of the function in question. It is possible to determine on the top level of `Result` if the function was successful, skipped (by the user), or if it failed.

It is also possible to dig deeper and, for example, perform pattern matching on the various reasons for skipped or failed. Notice that `{'EXIT',Reason}` tuples are translated into `{error,Reason}`. Notice also that if a `{failed,{Suite,end_per_testcase,FailInfo}}` result is received, the test case was successful, but `end_per_testcase` for the case failed.

`#event{name = tc_auto_skip, data = {Suite,TestName,Reason}}`

`Suite = atom()`, the name of the suite.

`TestName = init_per_suite | end_per_suite | {init_per_group,GroupName} | {end_per_group,GroupName} | {FuncName,GroupName} | FuncName`

`FuncName = atom()`, the name of the test case or configuration function.

`GroupName = atom()`, the name of the test case group.

`Reason = {failed,FailReason} | {require_failed_in_suite0,RequireInfo}`, reason for auto-skipping `Func`.

`FailReason = {Suite,ConfigFunc,FailInfo}} | {Suite,FailedCaseInSequence}`, reason for failure.

`RequireInfo = {not_available,atom() | tuple()}`, why require failed.

`ConfigFunc = init_per_suite | init_per_group`

`FailInfo = {timetrap_timeout,integer()} | {RunTimeError,StackTrace} | bad_return | UserTerm`, error details.

`FailedCaseInSequence = atom()`, the name of a case that failed in a sequence.

`RunTimeError = term()`, a runtime error, for example `badmatch` or `undef`.

`StackTrace = list()`, list of function calls preceeding a runtime error.

`UserTerm = term()`, any data specified by user, or `exit/1` information.

This event is sent for every test case or configuration function that `Common Test` has skipped automatically because of either a failed `init_per_suite` or `init_per_group`, a failed `require` in `suite/0`, or a failed test case in a sequence. Notice that this event is never received as a result of a test case getting skipped because of `init_per_testcase` failing, as that information is carried with event `tc_done`. If a failed test case belongs to a test case group, the second data element is a tuple `{FuncName,GroupName}`, otherwise only the function name.

`#event{name = tc_user_skip, data = {Suite,TestName,Comment}}`

`Suite = atom()`, the name of the suite.

`TestName = init_per_suite | end_per_suite | {init_per_group,GroupName} | {end_per_group,GroupName} | {FuncName,GroupName} | FuncName`

`FuncName = atom()`, the name of the test case or configuration function.

`GroupName = atom()`, the name of the test case group.

`Comment = string()`, why the test case was skipped.

This event specifies that a test case was skipped by the user. It is only received if the skip is declared in a test specification. Otherwise, user skip information is received as a `{skipped,SkipReason}` result in event `tc_done` for the test case. If a skipped test case belongs to a test case group, the second data element is a tuple `{FuncName,GroupName}`, otherwise only the function name.

`#event{name = test_stats, data = {Ok,Failed,Skipped}}`

`Ok = integer()`, current number of successful test cases.

`Failed = integer()`, current number of failed test cases.

`Skipped = {UserSkipped,AutoSkipped}`

`UserSkipped = integer()`, current number of user-skipped test cases.

`AutoSkipped = integer()`, current number of auto-skipped test cases.

This is a statistics event with current count of successful, skipped, and failed test cases so far. This event is sent after the end of each test case, immediately following event `tc_done`.

Internal Events

The internal events are as follows:

`#event{name = start_make, data = Dir}`

`Dir = string()`, running make in this directory.

This internal event says that `Common Test` starts compiling modules in directory `Dir`.

`#event{name = finished_make, data = Dir}`

`Dir = string()`, finished running make in this directory.

This internal event says that `Common Test` is finished compiling modules in directory `Dir`.

`#event{name = start_write_file, data = FullNameFile}`

`FullNameFile = string(), full name of the file.`

This internal event is used by the `Common Test` Master process to synchronize particular file operations.

`#event{name = finished_write_file, data = FullNameFile}`

`FullNameFile = string(), full name of the file.`

This internal event is used by the `Common Test` Master process to synchronize particular file operations.

Notes

The events are also documented in `ct_event.erl`. This module can serve as an example of what an event handler for the `Common Test` event manager can look like.

Note:

To ensure that printouts to `stdout` (or printouts made with *`ct:log/2,3`* or *`ct:pal,2,3`*) get written to the test case log file, and not to the `Common Test` framework log, you can synchronize with the `Common Test` server by matching on evvents `tc_start` and `tc_done`. In the period between these events, all I/O is directed to the test case log file. These events are sent synchronously to avoid potential timing problems (for example, that the test case log file is closed just before an I/O message from an external process gets through). Knowing this, you need to be careful that your `handle_event/2` callback function does not stall the test execution, possibly causing unexpected behavior as a result.

1.13 Dependencies between Test Cases and Suites

1.13.1 General

When creating test suites, it is strongly recommended to not create dependencies between test cases, that is, letting test cases depend on the result of previous test cases. There are various reasons for this, such as, the following:

- It makes it impossible to run test cases individually.
- It makes it impossible to run test cases in a different order.
- It makes debugging difficult (as a fault can be the result of a problem in a different test case than the one failing).
- There are no good and explicit ways to declare dependencies, so it can be difficult to see and understand these in test suite code and in test logs.
- Extending, restructuring, and maintaining test suites with test case dependencies is difficult.

There are often sufficient means to work around the need for test case dependencies. Generally, the problem is related to the state of the System Under Test (SUT). The action of one test case can change the system state. For some other test case to run properly, this new state must be known.

Instead of passing data between test cases, it is recommended that the test cases read the state from the SUT and perform assertions (that is, let the test case run if the state is as expected, otherwise reset or fail). It is also recommended to use the state to set variables necessary for the test case to execute properly. Common actions can often be implemented as library functions for test cases to call to set the SUT in a required state. (Such common actions can also be separately tested, if necessary, to ensure that they work as expected). It is sometimes also possible, but not always desirable, to group tests together in one test case, that is, let a test case perform a "scenario" test (a test consisting of subtests).

Consider, for example, a server application under test. The following functionality is to be tested:

- Starting the server
- Configuring the server

- Connecting a client to the server
- Disconnecting a client from the server
- Stopping the server

There are obvious dependencies between the listed functions. The server cannot be configured if it has not first been started, a client connot be connectd until the server is properly configured, and so on. If we want to have one test case for each function, we might be tempted to try to always run the test cases in the stated order and carry possible data (identities, handles, and so on) between the cases and therefore introduce dependencies between them.

To avoid this, we can consider starting and stopping the server for every test. We can thus implement the start and stop action as common functions to be called from `init_per_testcase` and `end_per_testcase`. (Remember to test the start and stop functionality separately.) The configuration can also be implemented as a common function, maybe grouped with the start function. Finally, the testing of connecting and disconnecting a client can be grouped into one test case. The resulting suite can look as follows:

```
-module(my_server_SUITE).
-compile(export_all).
-include_lib("ct.hrl").

%%% init and end functions...

suite() -> [{require,my_server_cfg}].

init_per_testcase(start_and_stop, Config) ->
    Config;

init_per_testcase(config, Config) ->
    [{server_pid,start_server()} | Config];

init_per_testcase(_, Config) ->
    ServerPid = start_server(),
    configure_server(),
    [{server_pid,ServerPid} | Config].

end_per_testcase(start_and_stop, _) ->
    ok;

end_per_testcase(_, _) ->
    ServerPid = ?config(server_pid),
    stop_server(ServerPid).

%%% test cases...

all() -> [start_and_stop, config, connect_and_disconnect].

%% test that starting and stopping works
start_and_stop(_) ->
    ServerPid = start_server(),
    stop_server(ServerPid).

%% configuration test
config(Config) ->
    ServerPid = ?config(server_pid, Config),
    configure_server(ServerPid).

%% test connecting and disconnecting client
connect_and_disconnect(Config) ->
    ServerPid = ?config(server_pid, Config),
    {ok,SessionId} = my_server:connect(ServerPid),
    ok = my_server:disconnect(ServerPid, SessionId).

%%% common functions...

start_server() ->
    {ok,ServerPid} = my_server:start(),
    ServerPid.

stop_server(ServerPid) ->
    ok = my_server:stop(),
    ok.

configure_server(ServerPid) ->
    ServerCfgData = ct:get_config(my_server_cfg),
    ok = my_server:configure(ServerPid, ServerCfgData),
    ok.
```

1.13.2 Saving Configuration Data

Sometimes it is impossible, or infeasible, to implement independent test cases. Maybe it is not possible to read the SUT state. Maybe resetting the SUT is impossible and it takes too long time to restart the system. In situations where test case dependency is necessary, CT offers a structured way to carry data from one test case to the next. The same mechanism can also be used to carry data from one test suite to the next.

The mechanism for passing data is called `save_config`. The idea is that one test case (or suite) can save the current value of `Config`, or any list of key-value tuples, so that the next executing test case (or test suite) can read it. The configuration data is not saved permanently but can only be passed from one case (or suite) to the next.

To save `Config` data, return tuple `{save_config,ConfigList}` from `end_per_testcase` or from the main test case function.

To read data saved by a previous test case, use macro `config` with a `saved_config` key as follows:

```
{Saver,ConfigList} = ?config(saved_config, Config)
```

`Saver` (`atom()`) is the name of the previous test case (where the data was saved). The `config` macro can be used to extract particular data also from the recalled `ConfigList`. It is strongly recommended that `Saver` is always matched to the expected name of the saving test case. This way, problems because of restructuring of the test suite can be avoided. Also, it makes the dependency more explicit and the test suite easier to read and maintain.

To pass data from one test suite to another, the same mechanism is used. The data is to be saved by finction *`end_per_suite`* and read by function *`init_per_suite`* in the suite that follows. When passing data between suites, `Saver` carries the name of the test suite.

Example:

```
-module(server_b_SUITE).
-compile(export_all).
-include_lib("ct.hrl").

%%% init and end functions...

init_per_suite(Config) ->
    %% read config saved by previous test suite
    {server_a_SUITE,OldConfig} = ?config(saved_config, Config),
    %% extract server identity (comes from server_a_SUITE)
    ServerId = ?config(server_id, OldConfig),
    SessionId = connect_to_server(ServerId),
    [{ids,{ServerId,SessionId}} | Config].

end_per_suite(Config) ->
    %% save config for server_c_SUITE (session_id and server_id)
    {save_config,Config}

%%% test cases...

all() -> [allocate, deallocate].

allocate(Config) ->
    {ServerId,SessionId} = ?config(ids, Config),
    {ok,Handle} = allocate_resource(ServerId, SessionId),
    %% save handle for deallocation test
    NewConfig = [{handle,Handle}],
    {save_config,NewConfig}.

deallocate(Config) ->
    {ServerId,SessionId} = ?config(ids, Config),
    {allocate,OldConfig} = ?config(saved_config, Config),
    Handle = ?config(handle, OldConfig),
    ok = deallocate_resource(ServerId, SessionId, Handle).
```

To save `Config` data from a test case that is to be skipped, return tuple `{skip_and_save,Reason,ConfigList}`.

The result is that the test case is skipped with `Reason` printed to the log file (as described earlier) and `ConfigList` is saved for the next test case. `ConfigList` can be read using `?config(saved_config, Config)`, as described earlier. `skip_and_save` can also be returned from `init_per_suite`. In this case, the saved data can be read by `init_per_suite` in the suite that follows.

1.13.3 Sequences

Sometimes test cases depend on each other so that if one case fails, the following tests are not to be executed. Typically, if the `save_config` facility is used and a test case that is expected to save data crashes, the following case cannot run. `Common Test` offers a way to declare such dependencies, called sequences.

A sequence of test cases is defined as a test case group with a `sequence` property. Test case groups are defined through function `groups/0` in the test suite (for details, see section *Test Case Groups*.

For example, to ensure that if `allocate` in `server_b_SUITE` crashes, `deallocate` is skipped, the following sequence can be defined:

```
groups() -> [{alloc_and_dealloc, [sequence], [alloc,dealloc]}].
```

Assume that the suite contains the test case `get_resource_status` that is independent of the other two cases, then function `all` can look as follows:

```
all() -> [{group,alloc_and_dealloc}, get_resource_status].
```

If `alloc` succeeds, `dealloc` is also executed. If `alloc` fails however, `dealloc` is not executed but marked as `SKIPPED` in the HTML log. `get_resource_status` runs no matter what happens to the `alloc_and_dealloc` cases.

Test cases in a sequence are executed in order until all succeed or one fails. If one fails, all following cases in the sequence are skipped. The cases in the sequence that have succeeded up to that point are reported as successful in the log. Any number of sequences can be specified.

Example:

```
groups() -> [{scenarioA, [sequence], [testA1, testA2]},
             {scenarioB, [sequence], [testB1, testB2, testB3]}].

all() -> [test1,
          test2,
          {group,scenarioA},
   test3,
          {group,scenarioB},
          test4].
```

A sequence group can have subgroups. Such subgroups can have any property, that is, they are not required to also be sequences. If you want the status of the subgroup to affect the sequence on the level above, return `{return_group_result,Status}` from *`end_per_group/2`*, as described in section *Repeated Groups* in Writing Test Suites. A failed subgroup (`Status == failed`) causes the execution of a sequence to fail in the same way a test case does.

1.14 Common Test Hooks

1.14.1 General

The **Common Test Hook (CTH)** framework allows extensions of the default behavior of `Common Test` using hooks before and after all test suite calls. CTHs allow advanced `Common Test` users to abstract out behavior that is common to multiple test suites without littering all test suites with library calls. This can be used for logging, starting, and monitoring external systems, building C files needed by the tests, and so on.

In brief, CTH allows you to do the following:

- Manipulate the runtime configuration before each suite configuration call.
- Manipulate the return of all suite configuration calls, and in extension, the result of the tests themselves.

The following sections describe how to use CTHs, when they are run, and how to manipulate the test results in a CTH.

Warning:

When executing within a CTH, all timetraps are shut off. So if your CTH never returns, the entire test run is stalled.

1.14.2 Installing a CTH

A CTH can be installed in multiple ways in your test run. You can do it for all tests in a run, for specific test suites, and for specific groups within a test suite. If you want a CTH to be present in all test suites within your test run, there are three ways to accomplish that, as follows:

- Add `-ct_hooks` as an argument to *ct_run*. To add multiple CTHs using this method, append them to each other using the keyword `and`, that is, `ct_run -ct_hooks cth1 [{debug,true}] and cth2 ....`
- Add tag `ct_hooks` to your *Test Specification*.
- Add tag `ct_hooks` to your call to *ct:run_test/1*.

CTHs can also be added within a test suite. This is done by returning `{ct_hooks,[CTH]}` in the configuration list from *suite/0*, *init_per_suite/1*, or *init_per_group/2*.

In this case, `CTH` can either be only the module name of the CTH or a tuple with the module name and the initial arguments, and optionally the hook priority of the CTH. For example, one of the following:

- `{ct_hooks,[my_cth_module]}`
- `{ct_hooks,[{my_cth_module,[{debug,true}]}]}`
- `{ct_hooks,[{my_cth_module,[{debug,true}],500}]}`

Overriding CTHs

By default, each installation of a CTH causes a new instance of it to be activated. This can cause problems if you want to override CTHs in test specifications while still having them in the suite information function. The *id/1* callback exists to address this problem. By returning the same `id` in both places, `Common Test` knows that this CTH is already installed and does not try to install it again.

CTH Execution Order

By default, each CTH installed is executed in the order that they are installed for init calls, and then reversed for end calls. This is not always desired, so `Common Test` allows the user to specify a priority for each hook. The priority can either be specified in the CTH function *init/2* or when installing the hook. The priority specified at installation overrides the priority returned by the CTH.

1.14.3 CTH Scope

Once the CTH is installed into a certain test run it remains there until its scope is expired. The scope of a CTH depends on when it is installed, see the following table. Function *init/2* is called at the beginning of the scope and function *terminate/1* is called when the scope ends.

CTH installed in	**CTH scope begins before**	**CTH scope ends after**
ct_run	the first test suite is to be run	the last test suite has been run
ct:run_test	the first test suite is run	the last test suite has been run
Test Specification	the first test suite is run	the last test suite has been run
suite/0	*pre_init_per_suite/3* is called	*post_end_per_suite/4* has been called for that test suite
init_per_suite/1	*post_init_per_suite/4* is called	*post_end_per_suite/4* has been called for that test suite
init_per_group/2	*post_init_per_group/5* is called	*post_end_per_group/5* has been called for that group

Table 14.1: Scope of a CTH

CTH Processes and Tables

CTHs are run with the same process scoping as normal test suites, that is, a different process executes the `init_per_suite` hooks then the `init_per_group` or `per_testcase` hooks. So if you want to spawn a process in the CTH, you cannot link with the CTH process, as it exits after the post hook ends. Also, if you for some reason need an ETS table with your CTH, you must spawn a process that handles it.

External Configuration Data and Logging

Configuration data values in the CTH can be read by calling *`ct:get_config/1,2,3`* (as explained in section *Requiring and Reading Configuration Data*). The configuration variables in question must, as always, first have been required by a suite-, group-, or test case information function, or by function *`ct:require/1/2`*. The latter can also be used in CT hook functions.

The CT hook functions can call any logging function in the `ct` interface to print information to the log files, or to add comments in the suite overview page.

1.14.4 Manipulating Tests

Through CTHs the results of tests and configuration functions can be manipulated. The main purpose to do this with CTHs is to allow common patterns to be abstracted out from test suites and applied to multiple test suites without duplicating any code. All the callback functions for a CTH follow a common interface described hereafter.

`Common Test` always calls all available hook functions, even pre- and post hooks for configuration functions that are not implemented in the suite. For example, `pre_init_per_suite(x_SUITE, ...)` and `post_init_per_suite(x_SUITE, ...)` are called for test suite `x_SUITE`, even if it does not export `init_per_suite/1`. With this feature hooks can be used as configuration fallbacks, and all configuration functions can be replaced with hook functions.

Pre Hooks

In a CTH, the behavior can be hooked in before the following functions:

- *`init_per_suite`*
- *`init_per_group`*
- *`init_per_testcase`*
- *`end_per_testcase`*
- *`end_per_group`*
- *`end_per_suite`*

This is done in the CTH functions called `pre_<name of function>`. These functions take the arguments `SuiteName`, `Name` (group or test case name, if applicable), `Config`, and `CTHState`. The return value of the CTH function is always a combination of a result for the suite/group/test and an updated `CTHState`.

To let the test suite continue on executing, return the configuration list that you want the test to use as the result.

All pre hooks, except `pre_end_per_testcase/4`, can skip or fail the test by returning a tuple with `skip` or `fail`, and a reason as the result.

Example:

```
pre_init_per_suite(SuiteName, Config, CTHState) ->
  case db:connect() of
    {error,_Reason} ->
      {{fail, "Could not connect to DB"}, CTHState};
    {ok, Handle} ->
      {[{db_handle, Handle} | Config], CTHState#state{ handle = Handle }}
  end.
```

Note:

If you use multiple CTHs, the first part of the return tuple is used as input for the next CTH. So in the previous example the next CTH can get `{fail,Reason}` as the second parameter. If you have many CTHs interacting, do not let each CTH return `fail` or `skip`. Instead, return that an action is to be taken through the `Config` list and implement a CTH that, at the end, takes the correct action.

Post Hooks

In a CTH, behavior can be hooked in after the following functions:

- *`init_per_suite`*
- *`init_per_group`*
- *`init_per_testcase`*
- *`end_per_testcase`*
- *`end_per_group`*
- *`end_per_suite`*

This is done in the CTH functions called `post_<name of function>`. These functions take the arguments `SuiteName`, `Name` (group or test case name, if applicable), `Config`, `Return`, and `CTHState`. `Config` in this case is the same `Config` as the testcase is called with. `Return` is the value returned by the testcase. If the testcase fails by crashing, `Return` is `{'EXIT',{{Error,Reason},Stacktrace}}`.

The return value of the CTH function is always a combination of a result for the suite/group/test and an updated `CTHState`. If you do not want the callback to affect the outcome of the test, return the `Return` data as it is given to the CTH. You can also modify the test result. By returning the `Config` list with element `tc_status` removed, you can recover from a test failure. As in all the pre hooks, it is also possible to fail/skip the test case in the post hook.

Example:

```
post_end_per_testcase(_Suite, _TC, Config, {'EXIT',{_,_}}, CTHState) ->
  case db:check_consistency() of
    true ->
      %% DB is good, pass the test.
      {proplists:delete(tc_status, Config), CTHState};
    false ->
      %% DB is not good, mark as skipped instead of failing
      {{skip, "DB is inconsisten!"}, CTHState}
  end;
post_end_per_testcase(_Suite, _TC, Config, Return, CTHState) ->
  %% Do nothing if tc does not crash.
  {Return, CTHState}.
```

Note:

Do recover from a testcase failure using CTHs only a last resort. If used wrongly, it can be very difficult to determine which tests that pass or fail in a test run.

Skip and Fail Hooks

After any post hook has been executed for all installed CTHs, *on_tc_fail* or *on_tc_skip* is called if the testcase failed or was skipped, respectively. You cannot affect the outcome of the tests any further at this point.

1.14.5 Synchronizing External User Applications with Common Test

CTHs can be used to synchronize test runs with external user applications. The init function can, for example, start and/or communicate with an application that has the purpose of preparing the SUT for an upcoming test run, or initialize a database for saving test data to during the test run. The terminate function can similarly order such an application to reset the SUT after the test run, and/or tell the application to finish active sessions and terminate. Any system error- or progress reports generated during the init- or termination stage are saved in the *Pre- and Post Test I/O Log*. (This is also true for any printouts made with `ct:log/2` and `ct:pal/2`).

To ensure that `Common Test` does not start executing tests, or closes its log files and shuts down, before the external application is ready for it, `Common Test` can be synchronized with the application. During startup and shutdown, `Common Test` can be suspended, simply by having a CTH evaluate a `receive` expression in the init- or terminate function. The macros `?CT_HOOK_INIT_PROCESS` (the process executing the hook init function) and `?CT_HOOK_TERMINATE_PROCESS` (the process executing the hook terminate function) each specifies the name of the correct `Common Test` process to send a message to. This is done to return from the `receive`. These macros are defined in `ct.hrl`.

1.14.6 Example CTH

The following CTH logs information about a test run into a format parseable by *file:consult/1* (in Kernel):

```
%%% Common Test Example Common Test Hook module.
-module(example_cth).

%% Callbacks
-export([id/1]).
-export([init/2]).

-export([pre_init_per_suite/3]).
-export([post_init_per_suite/4]).
-export([pre_end_per_suite/3]).
-export([post_end_per_suite/4]).

-export([pre_init_per_group/4]).
-export([post_init_per_group/5]).
-export([pre_end_per_group/4]).
-export([post_end_per_group/5]).

-export([pre_init_per_testcase/4]).
-export([post_init_per_testcase/5]).
-export([pre_end_per_testcase/4]).
-export([post_end_per_testcase/5]).

-export([on_tc_fail/4]).
-export([on_tc_skip/4]).

-export([terminate/1]).

-record(state, { file_handle, total, suite_total, ts, tcs, data }).

%% Return a unique id for this CTH.
id(Opts) ->
  proplists:get_value(filename, Opts, "/tmp/file.log").

%% Always called before any other callback function. Use this to initiate
%% any common state.
init(Id, Opts) ->
    {ok,D} = file:open(Id,[write]),
    {ok, #state{ file_handle = D, total = 0, data = [] }}.

%% Called before init_per_suite is called.
pre_init_per_suite(Suite,Config,State) ->
    {Config, State#state{ suite_total = 0, tcs = [] }}.

%% Called after init_per_suite.
post_init_per_suite(Suite,Config,Return,State) ->
    {Return, State}.

%% Called before end_per_suite.
pre_end_per_suite(Suite,Config,State) ->
    {Config, State}.

%% Called after end_per_suite.
post_end_per_suite(Suite,Config,Return,State) ->
    Data = {suites, Suite, State#state.suite_total, lists:reverse(State#state.tcs)},
    {Return, State#state{ data = [Data | State#state.data] ,
                          total = State#state.total + State#state.suite_total } }.

%% Called before each init_per_group.
pre_init_per_group(Suite,Group,Config,State) ->
    {Config, State}.

%% Called after each init_per_group.
post_init_per_group(Suite,Group,Config,Return,State) ->
    {Return, State}.
```

```
 %% Called before each end_per_group.
 pre_end_per_group(Suite,Group,Config,State) ->
     {Config, State}.

 %% Called after each end_per_group.
 post_end_per_group(Suite,Group,Config,Return,State) ->
     {Return, State}.

 %% Called before each init_per_testcase.
 pre_init_per_testcase(Suite,TC,Config,State) ->
     {Config, State#state{ ts = now(), total = State#state.suite_total + 1 } }.

 %% Called after each init_per_testcase (immediately before the test case).
 post_init_per_testcase(Suite,TC,Config,Return,State) ->
     {Return, State}

%% Called before each end_per_testcase (immediately after the test case).
 pre_end_per_testcase(Suite,TC,Config,State) ->
     {Config, State}.

 %% Called after each end_per_testcase.
 post_end_per_testcase(Suite,TC,Config,Return,State) ->
     TCInfo = {testcase, Suite, TC, Return, timer:now_diff(now(), State#state.ts)},
     {Return, State#state{ ts = undefined, tcs = [TCInfo | State#state.tcs] } }.

 %% Called after post_init_per_suite, post_end_per_suite, post_init_per_group,
 %% post_end_per_group and post_end_per_testcase if the suite, group or test case failed.
 on_tc_fail(Suite, TC, Reason, State) ->
     State.

 %% Called when a test case is skipped by either user action
 %% or due to an init function failing.
 on_tc_skip(Suite, TC, Reason, State) ->
     State.

 %% Called when the scope of the CTH is done
 terminate(State) ->
     io:format(State#state.file_handle, "~p.~n",
               [{test_run, State#state.total, State#state.data}]),
     file:close(State#state.file_handle),
     ok.
```

1.14.7 Built-In CTHs

`Common Test` is delivered with some general-purpose CTHs that can be enabled by the user to provide generic testing functionality. Some of these CTHs are enabled by default when `common_test` is started to run. They can be disabled by setting `enable_builtin_hooks` to `false` on the command line or in the test specification. The following two CTHs are delivered with `Common Test`:

`cth_log_redirect`

Built-in

Captures all log events that would normally be printed by the default logger handler, and prints them to the current test case log. If an event cannot be associated with a test case, it is printed in the `Common Test` framework log. This happens for test cases running in parallel and events occuring in-between test cases. You can configure the level of *SASL* reports using the normal SASL mechanisms.

`cth_surefire`

Not built-in

Captures all test results and outputs them as surefire XML into a file. The created file is by default called `junit_report.xml`. The file name can be changed by setting option `path` for this hook, for example:

```
-ct_hooks cth_surefire [{path,"/tmp/report.xml"}]
```

If option `url_base` is set, an extra attribute named `url` is added to each `testsuite` and `testcase` XML element. The value is constructed from `url_base` and a relative path to the test suite or test case log, respectively, for example:

```
-ct_hooks cth_surefire [{url_base, "http://myserver.com/"}]
```

gives an URL attribute value similar to

```
"http://myserver.com/ct_run.ct@myhost.2012-12-12_11.19.39/  x86_64-unknown-
linux-gnu.my_test.logs/run.2012-12-12_11.19.39/suite.log.html"
```

Surefire XML can, for example, be used by Jenkins to display test results.

1.15 Some Thoughts about Testing

1.15.1 Goals

It is not possible to prove that a program is correct by testing. On the contrary, it has been formally proven that it is impossible to prove programs in general by testing. Theoretical program proofs or plain examination of code can be viable options for those wishing to certify that a program is correct. The test server, as it is based on testing, cannot be used for certification. Its intended use is instead to (cost effectively) **find bugs**. A successful test suite is one that reveals a bug. If a test suite results in OK, then we know very little that we did not know before.

1.15.2 What to Test

There are many kinds of test suites. Some concentrate on calling every function or command (in the documented way) in a certain interface. Some others do the same, but use all kinds of illegal parameters, and verify that the server stays alive and rejects the requests with reasonable error codes. Some test suites simulate an application (typically consisting of a few modules of an application), some try to do tricky requests in general, and some test suites even test internal functions with help of special Load Modules on target.

Another interesting category of test suites is the one checking that fixed bugs do not reoccur. When a bugfix is introduced, a test case that checks for that specific bug is written and submitted to the affected test suites.

Aim for finding bugs. Write whatever test that has the highest probability of finding a bug, now or in the future. Concentrate more on the critical parts. Bugs in critical subsystems are much more expensive than others.

Aim for functionality testing rather than implementation details. Implementation details change quite often, and the test suites are to be long lived. Implementation details often differ on different platforms and versions. If implementation details must be tested, try to factor them out into separate test cases. These test cases can later be rewritten or skipped.

Also, aim for testing everything once, no less, no more. It is not effective having every test case fail only because one function in the interface changed.

2 Reference Manual

common_test

Erlang module

The Common Test framework is an environment for implementing and performing automatic and semi-automatic execution of test cases.

In brief, Common Test supports:

- Automated execution of test suites (sets of test cases)
- Logging of events during execution
- HTML presentation of test suite results
- HTML presentation of test suite code
- Support functions for test suite authors
- Step-by-step execution of test cases

The following section describes the mandatory and optional test suite functions that Common Test calls during test execution. For more details, see section *Writing Test Suites* in the User's Guide.

Test Case Callback Functions

The following functions define the callback interface for a test suite.

Exports

Module:all() -> Tests | {skip,Reason}

Types:

Tests = [TestCase | {testcase,TestCase,TCRepeatProps} | {group,GroupName} | {group,GroupName,Properties} | {group,GroupName,Properties,SubGroups}]

TestCase = atom()

TCRepeatProps = [{repeat,N} | {repeat_until_ok,N} | {repeat_until_fail,N}]

GroupName = atom()

Properties = [parallel | sequence | Shuffle | {GroupRepeatType,N}] | default

SubGroups = [{GroupName,Properties} | {GroupName,Properties,SubGroups}]

Shuffle = shuffle | {shuffle,Seed}

Seed = {integer(),integer(),integer()}

GroupRepeatType = repeat | repeat_until_all_ok | repeat_until_all_fail | repeat_until_any_ok | repeat_until_any_fail

N = integer() | forever

Reason = term()

MANDATORY

Returns the list of all test cases and test case groups in the test suite module to be executed. This list also specifies the order the cases and groups are executed by Common Test. A test case is represented by an atom, the name of the test case function, or a testcase tuple indicating that the test case shall be repeated. A test case group is represented by a group tuple, where GroupName, an atom, is the name of the group (defined in *groups/0*). Execution properties for groups can also be specified, both for a top-level group and for any of its subgroups. Group execution properties

specified here override properties in the group definition (see *groups/0*). (With value `default`, the group definition properties are used).

If `{skip,Reason}` is returned, all test cases in the module are skipped and `Reason` is printed on the HTML result page.

For details on groups, see section *Test Case Groups* in the User's Guide.

`Module:groups() -> GroupDefs`

Types:

```
GroupDefs = [Group]
Group = {GroupName,Properties,GroupsAndTestCases}
GroupName = atom()
Properties = [parallel | sequence | Shuffle | {GroupRepeatType,N}]
GroupsAndTestCases = [Group | {group,GroupName} | TestCase |
{testcase,TestCase,TCRepeatProps}]
TestCase = atom()
TCRepeatProps = [{repeat,N} | {repeat_until_ok,N} | {repeat_until_fail,N}]
Shuffle = shuffle | {shuffle,Seed}
Seed = {integer(),integer(),integer()}
GroupRepeatType = repeat | repeat_until_all_ok | repeat_until_all_fail |
repeat_until_any_ok | repeat_until_any_fail
N = integer() | forever
```

OPTIONAL

Defines test case groups. For details, see section *Test Case Groups* in the User's Guide.

`Module:suite() -> [Info]`

Types:

```
Info = {timetrap,Time} | {require,Required} | {require,Name,Required} |
{userdata,UserData} | {silent_connections,Conns} | {stylesheet,CSSFile} |
{ct_hooks, CTHs}
Time = TimeVal | TimeFunc
TimeVal = MilliSec | {seconds,integer()} | {minutes,integer()} |
{hours,integer()}
TimeFunc = {Mod,Func,Args} | Fun
MilliSec = integer()
Mod = atom()
Func = atom()
Args = list()
Fun = fun()
Required = Key | {Key,SubKeys} | {Key,SubKey} | {Key,SubKey,SubKeys}
Key = atom()
SubKeys = SubKey | [SubKey]
SubKey = atom()
Name = atom()
UserData = term()
```

```
Conns = [atom()]
CSSFile = string()
CTHs = [CTHModule |
        {CTHModule, CTHInitArgs} |
        {CTHModule, CTHInitArgs, CTHPriority}]
CTHModule = atom()
CTHInitArgs = term()
```

OPTIONAL

The test suite information function. Returns a list of tagged tuples specifying various properties related to the execution of this test suite (common for all test cases in the suite).

Tag `timetrap` sets the maximum time that each test case is allowed to execute (including *`init_per_testcase/2`* and *`end_per_testcase/2`*). If the timetrap time is exceeded, the test case fails with reason `timetrap_timeout`. A `TimeFunc` function can be used to set a new timetrap by returning a `TimeVal`. It can also be used to trigger a timetrap time-out by, at some point, returning a value other than a `TimeVal`. For details, see section *Timetrap Time-Outs* in the User's Guide.

Tag `require` specifies configuration variables required by test cases (or configuration functions) in the suite. If the required configuration variables are not found in any of the configuration files, all test cases are skipped. For details about the `require` functionality, see funtion *`ct:require/1,2`*.

With `userdata`, the user can specify any test suite-related information, which can be read by calling *`ct:userdata/2`*.

Tag `ct_hooks` specifies the *Common Test Hooks* to be run with this suite.

Other tuples than the ones defined are ignored.

For details about the test suite information function, see section *Test Suite Information Function* in the User's Guide.

```
Module:init_per_suite(Config) -> NewConfig | {skip,Reason} |
{skip_and_save,Reason,SaveConfig}
```

Types:

```
Config = NewConfig = SaveConfig = [{Key,Value}]
Key = atom()
Value = term()
Reason = term()
```

OPTIONAL; if this function is defined, then *`end_per_suite/1`* must also be defined.

This configuration function is called as the first function in the suite. It typically contains initializations that are common for all test cases in the suite, and that must only be done once. Parameter `Config` is the configuration data that can be modified. Whatever is returned from this function is specified as `Config` to all configuration functions and test cases in the suite.

If `{skip,Reason}` is returned, all test cases in the suite are skipped and `Reason` is printed in the overview log for the suite.

For information on `save_config` and `skip_and_save`, see section *Saving Configuration Data* in the User's Guide.

```
Module:end_per_suite(Config) -> term() | {save_config,SaveConfig}
```

Types:

```
Config = SaveConfig = [{Key,Value}]
```

```
Key = atom()
Value = term()
```

OPTIONAL; if this function is defined, then *`init_per_suite/1`* must also be defined.

This function is called as the last test case in the suite. It is meant to be used for cleaning up after *`init_per_suite/1`*.

For information on `save_config`, see section *Saving Configuration Data* in the User's Guide.

`Module:group(GroupName) -> [Info]`

Types:

```
Info = {timetrap,Time} | {require,Required} | {require,Name,Required} |
{userdata,UserData} | {silent_connections,Conns} | {stylesheet,CSSFile} |
{ct_hooks, CTHs}
Time = TimeVal | TimeFunc
TimeVal = MilliSec | {seconds,integer()} | {minutes,integer()} |
{hours,integer()}
TimeFunc = {Mod,Func,Args} | Fun
MilliSec = integer()
Mod = atom()
Func = atom()
Args = list()
Fun = fun()
Required = Key | {Key,SubKeys} | {Key,Subkey} | {Key,Subkey,SubKeys}
Key = atom()
SubKeys = SubKey | [SubKey]
SubKey = atom()
Name = atom()
UserData = term()
Conns = [atom()]
CSSFile = string()
CTHs = [CTHModule |
        {CTHModule, CTHInitArgs} |
        {CTHModule, CTHInitArgs, CTHPriority}]
CTHModule = atom()
CTHInitArgs = term()
```

OPTIONAL

The test case group information function. It is supposed to return a list of tagged tuples that specify various properties related to the execution of a test case group (that is, its test cases and subgroups). Properties set by *`group/1`* override properties with the same key that have been set previously by *`suite/0`*.

Tag `timetrap` sets the maximum time that each test case is allowed to execute (including *`init_per_testcase/2`* and *`end_per_testcase/2`*). If the timetrap time is exceeded, the test case fails with reason `timetrap_timeout`. A `TimeFunc` function can be used to set a new timetrap by returning a `TimeVal`. It can also be used to trigger a timetrap time-out by, at some point, returning a value other than a `TimeVal`. For details, see section *Timetrap Time-Outs* in the User's Guide.

Tag `require` specifies configuration variables required by test cases (or configuration functions) in the suite. If the required configuration variables are not found in any of the configuration files, all test cases in this group are skipped. For details about the `require` functionality, see function *`ct:require/1,2`*.

With `userdata`, the user can specify any test case group related information that can be read by calling *`ct:userdata/2`*.

Tag `ct_hooks` specifies the *Common Test Hooks* to be run with this suite.

Other tuples than the ones defined are ignored.

For details about the test case group information function, see section *Group Information Function* in the User's Guide.

```
Module:init_per_group(GroupName, Config) -> NewConfig | {skip,Reason}
```

Types:

`GroupName = atom()`
`Config = NewConfig = [{Key,Value}]`
`Key = atom()`
`Value = term()`
`Reason = term()`

OPTIONAL; if this function is defined, then *`end_per_group/2`* must also be defined.

This configuration function is called before execution of a test case group. It typically contains initializations that are common for all test cases and subgroups in the group, and that must only be performed once. `GroupName` is the name of the group, as specified in the group definition (see *`groups/0`*). Parameter `Config` is the configuration data that can be modified. The return value of this function is given as `Config` to all test cases and subgroups in the group.

If `{skip,Reason}` is returned, all test cases in the group are skipped and `Reason` is printed in the overview log for the group.

For information about test case groups, see section *Test Case Groups* in the User's Guide.

```
Module:end_per_group(GroupName, Config) -> term() |
{return_group_result,Status}
```

Types:

`GroupName = atom()`
`Config = [{Key,Value}]`
`Key = atom()`
`Value = term()`
`Status = ok | skipped | failed`

OPTIONAL; if this function is defined, then *`init_per_group/2`* must also be defined.

This function is called after the execution of a test case group is finished. It is meant to be used for cleaning up after *`init_per_group/2`*. A status value for a nested subgroup can be returned with `{return_group_result,Status}`. The status can be retrieved in *`end_per_group/2`* for the group on the level above. The status is also used by `Common Test` for deciding if execution of a group is to proceed if property `sequence` or `repeat_until_*` is set.

For details about test case groups, see section *Test Case Groups* in the User's Guide.

```
Module:init_per_testcase(TestCase, Config) -> NewConfig | {fail,Reason} |
{skip,Reason}
```

Types:

```
TestCase = atom()
Config = NewConfig = [{Key,Value}]
Key = atom()
Value = term()
Reason = term()
```

OPTIONAL; if this function is defined, then *`end_per_testcase/2`* must also be defined.

This function is called before each test case. Argument `TestCase` is the test case name, and `Config` (list of key-value tuples) is the configuration data that can be modified. The `NewConfig` list returned from this function is given as `Config` to the test case. If `{fail,Reason}` is returned, the test case is marked as failed without being executed.

If `{skip,Reason}` is returned, the test case is skipped and `Reason` is printed in the overview log for the suite.

`Module:end_per_testcase(TestCase, Config) -> term() | {fail,Reason} | {save_config,SaveConfig}`

Types:

```
TestCase = atom()
Config = SaveConfig = [{Key,Value}]
Key = atom()
Value = term()
Reason = term()
```

OPTIONAL; if this function is defined, then *`init_per_testcase/2`* must also be defined.

This function is called after each test case, and can be used to clean up after *`init_per_testcase/2`* and the test case. Any return value (besides `{fail,Reason}` and `{save_config,SaveConfig}`) is ignored. By returning `{fail,Reason}`, `TestCase` is marked as faulty (even though it was successful in the sense that it returned a value instead of terminating).

For information on `save_config`, see section *Saving Configuration Data* in the User's Guide.

`Module:Testcase() -> [Info]`

Types:

```
Info = {timetrap,Time} | {require,Required} | {require,Name,Required} |
{userdata,UserData} | {silent_connections,Conns}
Time = TimeVal | TimeFunc
TimeVal = MilliSec | {seconds,integer()} | {minutes,integer()} |
{hours,integer()}
TimeFunc = {Mod,Func,Args} | Fun
MilliSec = integer()
Mod = atom()
Func = atom()
Args = list()
Fun = fun()
Required = Key | {Key,SubKeys} | {Key,Subkey} | {Key,Subkey,SubKeys}
Key = atom()
SubKeys = SubKey | [SubKey]
SubKey = atom()
Name = atom()
```

UserData = term()
Conns = [atom()]

OPTIONAL

The test case information function. It is supposed to return a list of tagged tuples that specify various properties related to the execution of this particular test case. Properties set by *Testcase/0* override properties set previously for the test case by *group/1* or *suite/0*.

Tag timetrap sets the maximum time that the test case is allowed to execute. If the timetrap time is exceeded, the test case fails with reason timetrap_timeout. *init_per_testcase/2* and *end_per_testcase/2* are included in the timetrap time. A TimeFunc function can be used to set a new timetrap by returning a TimeVal. It can also be used to trigger a timetrap time-out by, at some point, returning a value other than a TimeVal. For details, see section *Timetrap Time-Outs* in the User's Guide.

Tag require specifies configuration variables that are required by the test case (or init_per_testcase/2 or end_per_testcase/2). If the required configuration variables are not found in any of the configuration files, the test case is skipped. For details about the require functionality, see function *ct:require/1,2*.

If timetrap or require is not set, the default values specified by *suite/0* (or *group/1*) are used.

With userdata, the user can specify any test case-related information that can be read by calling *ct:userdata/3*.

Other tuples than the ones defined are ignored.

For details about the test case information function, see section *Test Case Information Function* in the User's Guide.

```
Module:Testcase(Config) -> term() | {skip,Reason} | {comment,Comment} |
{save_config,SaveConfig} | {skip_and_save,Reason,SaveConfig} | exit()
```

Types:

Config = SaveConfig = [{Key,Value}]
Key = atom()
Value = term()
Reason = term()
Comment = string()

MANDATORY

The implementation of a test case. Call the functions to test and check the result. If something fails, ensure the function causes a runtime error or call *ct:fail/1,2* (which also causes the test case process to terminate).

Elements from the Config list can, for example, be read with proplists:get_value/2 in STDLIB (or the macro ?config defined in ct.hrl).

If you decide not to run the test case after all, return {skip,Reason}. Reason is then printed in field Comment on the HTML result page.

To print some information in field Comment on the HTML result page, return {comment,Comment}.

If the function returns anything else, the test case is considered successful. The return value always gets printed in the test case log file.

For details about test case implementation, see section *Test Cases* in the User's Guide.

For information on save_config and skip_and_save, see section *Saving Configuration Data* in the User's Guide.

ct_run

Command

The `ct_run` program is automatically installed with Erlang/OTP and the `Common Test` application (for more information, see section *Installation* in the User's Guide). The program accepts different start flags. Some flags trigger `ct_run` to start `Common Test` and pass on data to it. Some flags start an Erlang node prepared for running `Common Test` in a particular mode.

The interface function *`ct:run_test/1`*, corresponding to the `ct_run` program, is used for starting `Common Test` from the Erlang shell (or an Erlang program). For details, see the *`ct`* manual page.

`ct_run` also accepts Erlang emulator flags. These are used when `ct_run` calls `erl` to start the Erlang node (this makes it possible to add directories to the code server path, change the cookie on the node, start more applications, and so on).

With the optional flag `-erl_args`, options on the `ct_run` command line can be divided into two groups:

- One group that `Common Test` is to process (those preceding `-erl_args`).
- One group that `Common Test` is to ignore and pass on directly to the emulator (those following `-erl_args`).

Options preceding `-erl_args` that `Common Test` does not recognize are also passed on to the emulator untouched. By `-erl_args` the user can specify flags with the same name, but with different destinations, on the `ct_run` command line.

If flags `-pa` or `-pz` are specified in the `Common Test` group of options (preceding `-erl_args`), relative directories are converted to absolute and reinserted into the code path by `Common Test`. This is to avoid problems loading user modules when `Common Test` changes working directory during test runs. However, `Common Test` ignores flags `-pa` and `-pz` following `-erl_args` on the command line. These directories are added to the code path normally (that is, on specified form).

Exit status is set before the program ends. Value `0` indicates a successful test result, `1` indicates one or more failed or auto-skipped test cases, and `2` indicates test execution failure.

If `ct_run` is called with option `-help`, it prints all valid start flags to `stdout`.

Run Tests from Command Line

```
ct_run -dir TestDir1 TestDir2 .. TestDirN |
 [-dir TestDir] -suite Suite1 Suite2 .. SuiteN
 [-group Groups1 Groups2 .. GroupsN] [-case Case1 Case2 .. CaseN]
 [-step [config | keep_inactive]]
 [-config ConfigFile1 ConfigFile2 .. ConfigFileN]
 [-userconfig CallbackModule1 ConfigString1 and CallbackModule2
  ConfigString2 and .. CallbackModuleN ConfigStringN]
 [-decrypt_key Key] | [-decrypt_file KeyFile]
 [-label Label]
 [-logdir LogDir]
 [-logopts LogOpts]
 [-verbosity GenVLevel | [Category1 VLevel1 and
  Category2 VLevel2 and .. CategoryN VLevelN]]
 [-silent_connections [ConnType1 ConnType2 .. ConnTypeN]]
 [-stylesheet CSSFile]
 [-cover CoverCfgFile]
 [-cover_stop Bool]
 [-event_handler EvHandler1 EvHandler2 .. EvHandlerN] |
 [-event_handler_init EvHandler1 InitArg1 and
  EvHandler2 InitArg2 and .. EvHandlerN InitArgN]
 [-include InclDir1 InclDir2 .. InclDirN]
 [-no_auto_compile]
 [-abort_if_missing_suites]
 [-muliply_timetraps Multiplier]
 [-scale_timetraps]
 [-create_priv_dir auto_per_run | auto_per_tc | manual_per_tc]
 [-repeat N] |
 [-duration HHMMSS [-force_stop [skip_rest]]] |
 [-until [YYMoMoDD]HHMMSS [-force_stop [skip_rest]]]
 [-basic_html]
 [-no_esc_chars]
 [-keep_logs all | NLogs]
 [-ct_hooks CTHModule1 CTHOpts1 and CTHModule2 CTHOpts2 and ..
  CTHModuleN CTHOptsN]
 [-exit_status ignore_config]
 [-help]
```

Run Tests using Test Specification

```
ct_run -spec TestSpec1 TestSpec2 .. TestSpecN
 [-join_specs]
 [-config ConfigFile1 ConfigFile2 .. ConfigFileN]
 [-userconfig CallbackModule1 ConfigString1 and CallbackModule2
  ConfigString2 and .. and CallbackModuleN ConfigStringN]
 [-decrypt_key Key] | [-decrypt_file KeyFile]
 [-label Label]
 [-logdir LogDir]
 [-logopts LogOpts]
 [-verbosity GenVLevel | [Category1 VLevel1 and
  Category2 VLevel2 and .. CategoryN VLevelN]]
 [-allow_user_terms]
 [-silent_connections [ConnType1 ConnType2 .. ConnTypeN]]
 [-stylesheet CSSFile]
 [-cover CoverCfgFile]
 [-cover_stop Bool]
 [-event_handler EvHandler1 EvHandler2 .. EvHandlerN] |
 [-event_handler_init EvHandler1 InitArg1 and
  EvHandler2 InitArg2 and .. EvHandlerN InitArgN]
 [-include InclDir1 InclDir2 .. InclDirN]
 [-no_auto_compile]
 [-abort_if_missing_suites]
 [-muliply_timetraps Multiplier]
 [-scale_timetraps]
 [-create_priv_dir auto_per_run | auto_per_tc | manual_per_tc]
 [-repeat N] |
 [-duration HHMMSS [-force_stop [skip_rest]]] |
 [-until [YYMoMoDD]HHMMSS [-force_stop [skip_rest]]]
 [-basic_html]
 [-no_esc_chars]
 [-keep_logs all | NLogs]
 [-ct_hooks CTHModule1 CTHOpts1 and CTHModule2 CTHOpts2 and ..
  CTHModuleN CTHOptsN]
 [-exit_status ignore_config]
```

Run Tests in Web-Based GUI

```
ct_run -vts [-browser Browser]
 [-dir TestDir1 TestDir2 .. TestDirN] |
 [[dir TestDir] -suite Suite [[-group Group] [-case Case]]]
 [-config ConfigFile1 ConfigFile2 .. ConfigFileN]
 [-userconfig CallbackModule1 ConfigString1 and CallbackModule2
   ConfigString2 and .. and CallbackModuleN ConfigStringN]
 [-logopts LogOpts]
 [-verbosity GenVLevel | [Category1 VLevel1 and
  Category2 VLevel2 and .. CategoryN VLevelN]]
 [-decrypt_key Key] | [-decrypt_file KeyFile]
 [-include InclDir1 InclDir2 .. InclDirN]
 [-no_auto_compile]
 [-abort_if_missing_suites]
 [-muliply_timetraps Multiplier]
 [-scale_timetraps]
 [-create_priv_dir auto_per_run | auto_per_tc | manual_per_tc]
 [-basic_html]
 [-no_esc_chars]
 [-keep_logs all | NLogs]
```

Refresh HTML Index Files

```
ct_run -refresh_logs [-logdir LogDir] [-basic_html]
 [-keep_logs all | NLogs]
```

Run Common Test in Interactive Mode

```
ct_run -shell
 [-config ConfigFile1 ConfigFile2 ... ConfigFileN]
 [-userconfig CallbackModule1 ConfigString1 and CallbackModule2
  ConfigString2 and .. and CallbackModuleN ConfigStringN]
 [-decrypt_key Key] | [-decrypt_file KeyFile]
```

Start a Common Test Master Node

```
ct_run -ctmaster
```

See Also

For information about the start flags, see section *Running Tests and Analyzing Results* in the User's Guide.

ct

Erlang module

Main user interface for the `Common Test` framework.

This module implements the command-line interface for running tests and basic functions for `Common Test` case issues, such as configuration and logging.

Test Suite Support Macros

The `config` macro is defined in `ct.hrl`. This macro is to be used to retrieve information from the `Config` variable sent to all test cases. It is used with two arguments; the first is the name of the configuration variable to retrieve, the second is the `Config` variable supplied to the test case.

Possible configuration variables include:

- `data_dir` - Data file directory
- `priv_dir` - Scratch file directory
- Whatever added by *`init_per_suite/1`* or *`init_per_testcase/2`* in the test suite.

Data Types

```
handle() = pid()
```

The identity (handle) of a connection.

```
config_key() = atom()
```

A configuration key which exists in a configuration file

```
target_name() = atom()
```

A name and association to configuration data introduced through a require statement, or a call to *`ct:require/2`*, for example, `ct:require(mynodename,{node,[telnet]})`.

```
key_or_name() = config_key() | target_name()
conn_log_options() = [conn_log_option()]
```

Options that can be given to the `cth_conn_log` hook, which is used for logging of NETCONF and Telnet connections. See *ct_netconfc* or *ct_telnet* for description and examples of how to use this hook.

```
conn_log_option() = {log_type,conn_log_type()} | {hosts,[key_or_name()]}
conn_log_type() = raw | pretty | html | silent
conn_log_mod() = ct_netconfc | ct_telnet
```

Exports

```
abort_current_testcase(Reason) -> ok | {error, ErrorReason}
```

Types:

`Reason = term()`

`ErrorReason = no_testcase_running | parallel_group`

Aborts the currently executing test case. The user must know with certainty which test case is currently executing. The function is therefore only safe to call from a function that has been called (or synchronously invoked) by the test case.

`Reason`, the reason for aborting the test case, is printed in the test case log.

```
add_config(Callback, Config) -> ok | {error, Reason}
```

Types:

Callback = atom()

Config = string()

Reason = term()

Loads configuration variables using the specified callback module and configuration string. The callback module is to be either loaded or present in the code part. Loaded configuration variables can later be removed using function *ct:remove_config/2*.

```
break(Comment) -> ok | {error, Reason}
```

Types:

Comment = string()

Reason = {multiple_cases_running, TestCases} | 'enable break with release_shell option'

TestCases = [atom()]

Cancels any active timetrap and pauses the execution of the current test case until the user calls function `continue/0`. The user can then interact with the Erlang node running the tests, for example, for debugging purposes or for manually executing a part of the test case. If a parallel group is executing, *ct:break/2* is to be called instead.

A cancelled timetrap is not automatically reactivated after the break, but must be started exlicitly with *ct:timetrap/1*.

In order for the break/continue functionality to work, `Common Test` must release the shell process controlling `stdin`. This is done by setting start option `release_shell` to `true`. For details, see section *Running Tests from the Erlang Shell or from an Erlang Program* in the User's Guide.

```
break(TestCase, Comment) -> ok | {error, Reason}
```

Types:

TestCase = atom()

Comment = string()

Reason = 'test case not running' | 'enable break with release_shell option'

Works the same way as *ct:break/1*, only argument `TestCase` makes it possible to pause a test case executing in a parallel group. Function *ct:continue/1* is to be used to resume execution of `TestCase`.

For details, see *ct:break/1*.

```
capture_get() -> ListOfStrings
```

Types:

ListOfStrings = [string()]

Equivalent to *ct:capture_get([default])*.

```
capture_get(ExclCategories) -> ListOfStrings
```

Types:

ExclCategories = [atom()]

ListOfStrings = [string()]

Returns and purges the list of text strings buffered during the latest session of capturing printouts to `stdout`. Log categories that are to be ignored in `ListOfStrings` can be specified with `ExclCategories`. If `ExclCategories = []`, no filtering takes place.

See also *`ct:capture_start/0`*, *`ct:capture_stop/0`*, *`ct:log/3`*.

`capture_start() -> ok`

Starts capturing all text strings printed to `stdout` during execution of the test case.

See also *`ct:capture_get/1`*, *`ct:capture_stop/0`*.

`capture_stop() -> ok`

Stops capturing text strings (a session started with `capture_start/0`).

See also *`ct:capture_get/1`*, *`ct:capture_start/0`*.

`comment(Comment) -> ok`

Types:

`Comment = term()`

Prints the specified `Comment` in the comment field in the table on the test suite result page.

If called several times, only the last comment is printed. The test case return value `{comment,Comment}` overwrites the string set by this function.

`comment(Format, Args) -> ok`

Types:

`Format = string()`
`Args = list()`

Prints the formatted string in the comment field in the table on the test suite result page.

Arguments `Format` and `Args` are used in a call to `io_lib:format/2` to create the comment string. The behavior of `comment/2` is otherwise the same as function *`ct:comment/1`*.

`continue() -> ok`

This function must be called to continue after a test case (not executing in a parallel group) has called function *`ct:break/1`*.

`continue(TestCase) -> ok`

Types:

`TestCase = atom()`

This function must be called to continue after a test case has called *`ct:break/2`*. If the paused test case, `TestCase`, executes in a parallel group, this function, rather than `continue/0`, must be used to let the test case proceed.

`decrypt_config_file(EncryptFileName, TargetFileName) -> ok | {error, Reason}`

Types:

`EncryptFileName = string()`
`TargetFileName = string()`
`Reason = term()`

Decrypts `EncryptFileName`, previously generated with *`ct:encrypt_config_file/2,3`*. The original file contents is saved in the target file. The encryption key, a string, must be available in a text file named `.ct_config.crypt`, either in the current directory, or the home directory of the user (it is searched for in that order).

```
decrypt_config_file(EncryptFileName, TargetFileName, KeyOrFile) -> ok |
{error, Reason}
```

Types:

`EncryptFileName = string()`
`TargetFileName = string()`
`KeyOrFile = {key, string()} | {file, string()}`
`Reason = term()`

Decrypts `EncryptFileName`, previously generated with *`ct:encrypt_config_file/2,3`*. The original file contents is saved in the target file. The key must have the same value as that used for encryption.

```
encrypt_config_file(SrcFileName, EncryptFileName) -> ok | {error, Reason}
```

Types:

`SrcFileName = string()`
`EncryptFileName = string()`
`Reason = term()`

Encrypts the source configuration file with DES3 and saves the result in file `EncryptFileName`. The key, a string, must be available in a text file named `.ct_config.crypt`, either in the current directory, or the home directory of the user (it is searched for in that order).

For information about using encrypted configuration files when running tests, see section *Encrypted Configuration Files* in the User's Guide.

For details on DES3 encryption/decryption, see application *`Crypto`*.

```
encrypt_config_file(SrcFileName, EncryptFileName, KeyOrFile) -> ok | {error,
Reason}
```

Types:

`SrcFileName = string()`
`EncryptFileName = string()`
`KeyOrFile = {key, string()} | {file, string()}`
`Reason = term()`

Encrypts the source configuration file with DES3 and saves the result in the target file `EncryptFileName`. The encryption key to use is either the value in `{key,Key}` or the value stored in the file specified by `{file,File}`.

For information about using encrypted configuration files when running tests, see section *Encrypted Configuration Files* in the User's Guide.

For details on DES3 encryption/decryption, see application *`Crypto`*.

```
fail(Reason) -> ok
```

Types:

`Reason = term()`

Terminates a test case with the specified error `Reason`.

`fail(Format, Args) -> ok`

Types:

`Format = string()`
`Args = list()`

Terminates a test case with an error message specified by a format string and a list of values (used as arguments to `io_lib:format/2`).

`get_config(Required) -> Value`

Equivalent to *`ct:get_config(Required, undefined, [])`*.

`get_config(Required, Default) -> Value`

Equivalent to *`ct:get_config(Required, Default, [])`*.

`get_config(Required, Default, Opts) -> ValueOrElement`

Types:

`Required = KeyOrName | {KeyOrName, SubKey} | {KeyOrName, SubKey, SubKey}`
`KeyOrName = atom()`
`SubKey = atom()`
`Default = term()`
`Opts = [Opt] | []`
`Opt = element | all`
`ValueOrElement = term() | Default`

Reads configuration data values.

Returns the matching values or configuration elements, given a configuration variable key or its associated name (if one has been specified with *`ct:require/2`* or a `require` statement).

Example:

Given the following configuration file:

```
{unix,[{telnet,IpAddr},
       {user,[{username,Username},
              {password,Password}]}]}.
```

Then:

```
ct:get_config(unix,Default) -> [{telnet,IpAddr},
 {user, [{username,Username}, {password,Password}]}]
ct:get_config({unix,telnet},Default) -> IpAddr
ct:get_config({unix,user,username},Default) -> Username
ct:get_config({unix,ftp},Default) -> Default
ct:get_config(unknownkey,Default) -> Default
```

If a configuration variable key has been associated with a name (by *`ct:require/2`* or a `require` statement), the name can be used instead of the key to read the value:

```
ct:require(myuser,{unix,user}) -> ok.
ct:get_config(myuser,Default) -> [{username,Username}, {password,Password}]
```

If a configuration variable is defined in multiple files, use option `all` to access all possible values. The values are returned in a list. The order of the elements corresponds to the order that the configuration files were specified at startup.

If configuration elements (key-value tuples) are to be returned as result instead of values, use option `element`. The returned elements are then on the form `{Required,Value}`.

See also *`ct:get_config/1`*, *`ct:get_config/2`*, *`ct:require/1`*, *`ct:require/2`*.

`get_event_mgr_ref() -> EvMgrRef`

Types:

`EvMgrRef = atom()`

Gets a reference to the `Common Test` event manager. The reference can be used to, for example, add a user-specific event handler while tests are running.

Example:

```
gen_event:add_handler(ct:get_event_mgr_ref(), my_ev_h, [])
```

`get_progname() -> string()`

Returns the command used to start this Erlang instance. If this information could not be found, the string `"no_prog_name"` is returned.

`get_status() -> TestStatus | {error, Reason} | no_tests_running`

Types:

`TestStatus = [StatusElem]`
`StatusElem = {current, TestCaseInfo} | {successful, Successful} | {failed, Failed} | {skipped, Skipped} | {total, Total}`
`TestCaseInfo = {Suite, TestCase} | [{Suite, TestCase}]`
`Suite = atom()`
`TestCase = atom()`
`Successful = integer()`
`Failed = integer()`
`Skipped = {UserSkipped, AutoSkipped}`
`UserSkipped = integer()`
`AutoSkipped = integer()`
`Total = integer()`
`Reason = term()`

Returns status of ongoing test. The returned list contains information about which test case is executing (a list of cases when a parallel test case group is executing), as well as counters for successful, failed, skipped, and total test cases so far.

`get_target_name(Handle) -> {ok, TargetName} | {error, Reason}`

Types:

`Handle = handle()`
`TargetName = target_name()`

Returns the name of the target that the specified connection belongs to.

```
get_testspec_terms() -> TestSpecTerms | undefined
```

Types:

```
TestSpecTerms = [{Tag, Value}]
Value = [term()]
```

Gets a list of all test specification terms used to configure and run this test.

```
get_testspec_terms(Tags) -> TestSpecTerms | undefined
```

Types:

```
Tags = [Tag] | Tag
Tag = atom()
TestSpecTerms = [{Tag, Value}] | {Tag, Value}
Value = [{Node, term()}] | [term()]
Node = atom()
```

Reads one or more terms from the test specification used to configure and run this test. `Tag` is any valid test specification tag, for example, `label`, `config`, or `logdir`. User-specific terms are also available to read if option `allow_user_terms` is set.

All value tuples returned, except user terms, have the node name as first element.

To read test terms, use `Tag = tests` (rather than `suites`, `groups`, or `cases`). `Value` is then the list of **all** tests on the form `[{Node,Dir,[{TestSpec,GroupsAndCases1},...]},...]`, where `GroupsAndCases = [{Group,[Case]}] | [Case]`.

```
get_timetrap_info() -> {Time, {Scaling,ScaleVal}}
```

Types:

```
Time = integer() | infinity
Scaling = true | false
ScaleVal = integer()
```

Reads information about the timetrap set for the current test case. `Scaling` indicates if `Common Test` will attempt to compensate timetraps automatically for runtime delays introduced by, for example, tools like cover. `ScaleVal` is the value of the current scaling multipler (always 1 if scaling is disabled). Note the `Time` is not the scaled result.

```
get_verbosity(Category) -> Level | undefined
```

Types:

```
Category = default | atom()
Level = integer()
```

This function returns the verbosity level for the specified logging category. See the *User's Guide* for details. Use the value `default` to read the general verbosity level.

```
install(Opts) -> ok | {error, Reason}
```

Types:

```
Opts = [Opt]
Opt = {config, ConfigFiles} | {event_handler, Modules} | {decrypt,
KeyOrFile}
ConfigFiles = [ConfigFile]
ConfigFile = string()
```

Modules = [atom()]
KeyOrFile = {key, Key} | {file, KeyFile}
Key = string()
KeyFile = string()

Installs configuration files and event handlers.

Run this function once before the first test.

Example:

```
install([{config,["config_node.ctc","config_user.ctc"]}])
```

This function is automatically run by program `ct_run`.

`listenv(Telnet) -> [Env]`

Types:

Telnet = term()
Env = {Key, Value}
Key = string()
Value = string()

Performs command `listenv` on the specified Telnet connection and returns the result as a list of key-value pairs.

`log(Format) -> ok`

Equivalent to *`ct:log(default, 50, Format, [], [])`*.

`log(X1, X2) -> ok`

Types:

X1 = Category | Importance | Format
X2 = Format | FormatArgs

Equivalent to *`ct:log(Category, Importance, Format, FormatArgs, [])`*.

`log(X1, X2, X3) -> ok`

Types:

X1 = Category | Importance
X2 = Importance | Format
X3 = Format | FormatArgs | Opts

Equivalent to *`ct:log(Category, Importance, Format, FormatArgs, Opts)`*.

`log(X1, X2, X3, X4) -> ok`

Types:

X1 = Category | Importance
X2 = Importance | Format
X3 = Format | FormatArgs
X4 = FormatArgs | Opts

Equivalent to *`ct:log(Category, Importance, Format, FormatArgs, Opts)`*.

```
log(Category, Importance, Format, FormatArgs, Opts) -> ok
```

Types:

Category = atom()
Importance = integer()
Format = string()
FormatArgs = list()
Opts = [Opt]
Opt = {heading,string()} | no_css | esc_chars

Prints from a test case to the log file.

This function is meant for printing a string directly from a test case to the test case log file.

Default `Category` is `default`, default `Importance` is `?STD_IMPORTANCE`, and default value for `FormatArgs` is `[]`.

For details on `Category`, `Importance` and the `no_css` option, see section *Logging - Categories and Verbosity Levels* in the User's Guide.

Common Test will not escape special HTML characters (<, > and &) in the text printed with this function, unless the `esc_chars` option is used.

```
make_priv_dir() -> ok | {error, Reason}
```

Types:

Reason = term()

If the test is started with option `create_priv_dir` set to `manual_per_tc`, in order for the test case to use the private directory, it must first create it by calling this function.

```
notify(Name, Data) -> ok
```

Types:

Name = atom()
Data = term()

Sends an asynchronous notification of type `Name` with `Data`to the Common Test event manager. This can later be caught by any installed event manager.

See also *`gen_event(3)`*.

```
pal(Format) -> ok
```

Equivalent to *`ct:pal(default, 50, Format, [], [])`*.

```
pal(X1, X2) -> ok
```

Types:

X1 = Category | Importance | Format
X2 = Format | FormatArgs

Equivalent to *`ct:pal(Category, Importance, Format, FormatArgs, [])`*.

```
pal(X1, X2, X3) -> ok
```

Types:

X1 = Category | Importance

X2 = Importance | Format
X3 = Format | FormatArgs | Opts

Equivalent to *ct:pal(Category, Importance, Format, FormatArgs, Opts)*.

`pal(X1, X2, X3, X4) -> ok`

Types:

X1 = Category | Importance
X2 = Importance | Format
X3 = Format | FormatArgs
X4 = FormatArgs | Opts

Equivalent to *ct:pal(Category, Importance, Format, FormatArgs, Opts)*.

`pal(Category, Importance, Format, FormatArgs, Opts) -> ok`

Types:

Category = atom()
Importance = integer()
Format = string()
FormatArgs = list()
Opts = [Opt]
Opt = {heading,string()} | no_css

Prints and logs from a test case.

This function is meant for printing a string from a test case, both to the test case log file and to the console.

Default `Category` is `default`, default `Importance` is `?STD_IMPORTANCE`, and default value for `FormatArgs` is `[]`.

For details on `Category` and `Importance`, see section *Logging - Categories and Verbosity Levels* in the User's Guide.

Note that special characters in the text (<, > and &) will be escaped by Common Test before the text is printed to the log file.

`parse_table(Data) -> {Heading, Table}`

Types:

Data = [string()]
Heading = tuple()
Table = [tuple()]

Parses the printout from an SQL table and returns a list of tuples.

The printout to parse is typically the result of a `select` command in SQL. The returned `Table` is a list of tuples, where each tuple is a row in the table.

`Heading` is a tuple of strings representing the headings of each column in the table.

`print(Format) -> ok`

Equivalent to *ct:print(default, 50, Format, [], [])*.

`print(X1, X2) -> ok`

Types:

X1 = Category | Importance | Format
X2 = Format | FormatArgs

Equivalent to *`ct:print(Category, Importance, Format, FormatArgs, [])`*.

`print(X1, X2, X3) -> ok`

Types:

X1 = Category | Importance
X2 = Importance | Format
X3 = Format | FormatArgs | Opts

Equivalent to *`ct:print(Category, Importance, Format, FormatArgs, Opts)`*.

`print(X1, X2, X3, X4) -> ok`

Types:

X1 = Category | Importance
X2 = Importance | Format
X3 = Format | FormatArgs
X4 = FormatArgs | Opts

Equivalent to *`ct:print(Category, Importance, Format, FormatArgs, Opts)`*.

`print(Category, Importance, Format, FormatArgs, Opts) -> ok`

Types:

Category = atom()
Importance = integer()
Format = string()
FormatArgs = list()
Opts = [Opt]
Opt = {heading,string()}

Prints from a test case to the console.

This function is meant for printing a string from a test case to the console.

Default `Category` is `default`, default `Importance` is `?STD_IMPORTANCE`, and default value for `FormatArgs` is `[]`.

For details on `Category` and `Importance`, see section *Logging - Categories and Verbosity Levels* in the User's Guide.

`reload_config(Required) -> ValueOrElement | {error, Reason}`

Types:

Required = KeyOrName | {KeyOrName, SubKey} | {KeyOrName, SubKey, SubKey}
KeyOrName = atom()
SubKey = atom()
ValueOrElement = term()

Reloads configuration file containing specified configuration key.

This function updates the configuration data from which the specified configuration variable was read, and returns the (possibly) new value of this variable.

If some variables were present in the configuration, but are not loaded using this function, they are removed from the configuration table together with their aliases.

```
remaining_test_procs() -> {TestProcs,SharedGL,OtherGLs}
```

Types:

```
TestProcs = [{pid(),GL}]
GL = pid()
SharedGL = pid()
OtherGLs = [pid()]
```

This function will return the identity of test- and group leader processes that are still running at the time of this call. `TestProcs` are processes in the system that have a Common Test IO process as group leader. `SharedGL` is the central Common Test IO process, responsible for printing to log files for configuration functions and sequentially executing test cases. `OtherGLs` are Common Test IO processes that print to log files for test cases in parallel test case groups.

The process information returned by this function may be used to locate and terminate remaining processes after tests have finished executing. The function would typically by called from Common Test Hook functions.

Note that processes that execute configuration functions or test cases are never included in `TestProcs`. It is therefore safe to use post configuration hook functions (such as post_end_per_suite, post_end_per_group, post_end_per_testcase) to terminate all processes in `TestProcs` that have the current group leader process as its group leader.

Note also that the shared group leader (`SharedGL`) must never be terminated by the user, only by Common Test. Group leader processes for parallel test case groups (`OtherGLs`) may however be terminated in post_end_per_group hook functions.

```
remove_config(Callback, Config) -> ok
```

Types:

```
Callback = atom()
Config = string()
Reason = term()
```

Removes configuration variables (together wih their aliases) that were loaded with specified callback module and configuration string.

```
require(Required) -> ok | {error, Reason}
```

Types:

```
Required = Key | {Key, SubKeys} | {Key, SubKey, SubKeys}
Key = atom()
SubKeys = SubKey | [SubKey]
SubKey = atom()
```

Checks if the required configuration is available. Arbitrarily deep tuples can be specified as `Required`. Only the last element of the tuple can be a list of `SubKey`s.

Example 1. Require the variable `myvar`:

```
ok = ct:require(myvar).
```

In this case the configuration file must at least contain:

```
{myvar,Value}.
```

Example 2. Require key `myvar` with subkeys `sub1` and `sub2`:

```
ok = ct:require({myvar,[sub1,sub2]}).
```

In this case the configuration file must at least contain:

```
{myvar,[{sub1,Value},{sub2,Value}]}.
```

Example 3. Require key `myvar` with subkey `sub1` with `subsub1`:

```
ok = ct:require({myvar,sub1,sub2}).
```

In this case the configuration file must at least contain:

```
{myvar,[{sub1,[{sub2,Value}]}]}.
```

See also *`ct:get_config/1`*, *`ct:get_config/2`*, *`ct:get_config/3`*, *`ct:require/2`*.

`require(Name, Required) -> ok | {error, Reason}`

Types:

`Name = atom()`
`Required = Key | {Key, SubKey} | {Key, SubKey, SubKey}`
`SubKey = Key`
`Key = atom()`

Checks if the required configuration is available and gives it a name. The semantics for `Required` is the same as in *`ct:require/1`* except that a list of `SubKey`s cannot be specified.

If the requested data is available, the subentry is associated with `Name` so that the value of the element can be read with *`ct:get_config/1,2`* provided `Name` is used instead of the whole `Required` term.

Example:

Require one node with a Telnet connection and an FTP connection. Name the node `a`:

```
ok = ct:require(a,{machine,node}).
```

All references to this node can then use the node name. For example, a file over FTP is fetched like follows:

```
ok = ct:ftp_get(a,RemoteFile,LocalFile).
```

For this to work, the configuration file must at least contain:

```
{machine,[{node,[{telnet,IpAddr},{ftp,IpAddr}]}]}.
```

Note:

The behavior of this function changed radically in `Common Test` 1.6.2. To keep some backwards compatability, it is still possible to do:
`ct:require(a,{node,[telnet,ftp]}).`
This associates the name `a` with the top-level `node` entry. For this to work, the configuration file must at least contain:
`{node,[{telnet,IpAddr},{ftp,IpAddr}]}.`

See also *ct:get_config/1*, *ct:get_config/2*, *ct:get_config/3*, *ct:require/1*.

`run(TestDirs) -> Result`

Types:

`TestDirs = TestDir | [TestDir]`

Runs all test cases in all suites in the specified directories.

See also *ct:run/3*.

`run(TestDir, Suite) -> Result`

Runs all test cases in the specified suite.

See also *ct:run/3*.

`run(TestDir, Suite, Cases) -> Result`

Types:

`TestDir = string()`
`Suite = atom()`
`Cases = atom() | [atom()]`
`Result = [TestResult] | {error, Reason}`

Runs the specified test cases.

Requires that *ct:install/1* has been run first.

Suites (`*_SUITE.erl`) files must be stored in `TestDir` or `TestDir/test`. All suites are compiled when the test is run.

`run_test(Opts) -> Result`

Types:

`Opts = [OptTuples]`
`OptTuples = {dir, TestDirs} | {suite, Suites} | {group, Groups} | {testcase, Cases} | {spec, TestSpecs} | {join_specs, Bool} | {label, Label} | {config, CfgFiles} | {userconfig, UserConfig} | {allow_user_terms, Bool} | {logdir, LogDir} | {silent_connections, Conns} | {stylesheet, CSSFile} | {cover, CoverSpecFile} | {cover_stop, Bool} | {step, StepOpts} | {event_handler, EventHandlers} | {include, InclDirs} | {auto_compile, Bool} | {abort_if_missing_suites, Bool} | {create_priv_dir, CreatePrivDir} | {multiply_timetraps, M} | {scale_timetraps, Bool} | {repeat, N} | {duration, DurTime} | {until, StopTime} | {force_stop, ForceStop} | {decrypt, DecryptKeyOrFile} | {refresh_logs, LogDir}`

```
| {logopts, LogOpts} | {verbosity, VLevels} | {basic_html, Bool}
| {esc_chars, Bool} | {keep_logs,KeepSpec} | {ct_hooks, CTHs} |
{enable_builtin_hooks, Bool} | {release_shell, Bool}
TestDirs = [string()] | string()
Suites = [string()] | [atom()] | string() | atom()
Cases = [atom()] | atom()
Groups = GroupNameOrPath | [GroupNameOrPath]
GroupNameOrPath = [atom()] | atom() | all
TestSpecs = [string()] | string()
Label = string() | atom()
CfgFiles = [string()] | string()
UserConfig = [{CallbackMod, CfgStrings}] | {CallbackMod, CfgStrings}
CallbackMod = atom()
CfgStrings = [string()] | string()
LogDir = string()
Conns = all | [atom()]
CSSFile = string()
CoverSpecFile = string()
StepOpts = [StepOpt] | []
StepOpt = config | keep_inactive
EventHandlers = EH | [EH]
EH = atom() | {atom(), InitArgs} | {[atom()], InitArgs}
InitArgs = [term()]
InclDirs = [string()] | string()
CreatePrivDir = auto_per_run | auto_per_tc | manual_per_tc
M = integer()
N = integer()
DurTime = string(HHMMSS)
StopTime = string(YYMoMoDDHHMMSS) | string(HHMMSS)
ForceStop = skip_rest | Bool
DecryptKeyOrFile = {key, DecryptKey} | {file, DecryptFile}
DecryptKey = string()
DecryptFile = string()
LogOpts = [LogOpt]
LogOpt = no_nl | no_src
VLevels = VLevel | [{Category, VLevel}]
VLevel = integer()
Category = atom()
KeepSpec = all | pos_integer()
CTHs = [CTHModule | {CTHModule, CTHInitArgs}]
CTHModule = atom()
CTHInitArgs = term()
Result = {Ok, Failed, {UserSkipped, AutoSkipped}} | TestRunnerPid |
{error, Reason}
```

Ok = integer()
Failed = integer()
UserSkipped = integer()
AutoSkipped = integer()
TestRunnerPid = pid()
Reason = term()

Runs tests as specified by the combination of options in `Opts`. The options are the same as those used with program `ct_run`, see *Run Tests from Command Line* in the `ct_run` manual page.

Here a `TestDir` can be used to point out the path to a `Suite`. Option `testcase` corresponds to option `-case` in program `ct_run`. Configuration files specified in `Opts` are installed automatically at startup.

`TestRunnerPid` is returned if `release_shell == true`. For details, see *`ct:break/1`*.

`Reason` indicates the type of error encountered.

`run_testspec(TestSpec) -> Result`

Types:

TestSpec = [term()]
Result = {Ok, Failed, {UserSkipped, AutoSkipped}} | {error, Reason}
Ok = integer()
Failed = integer()
UserSkipped = integer()
AutoSkipped = integer()
Reason = term()

Runs a test specified by `TestSpec`. The same terms are used as in test specification files.

`Reason` indicates the type of error encountered.

`set_verbosity(Category, Level) -> ok`

Types:

Category = default | atom()
Level = integer()

Use this function to set, or modify, the verbosity level for a logging category. See the *User's Guide* for details. Use the value `default` to set the general verbosity level.

`sleep(Time) -> ok`

Types:

Time = {hours, Hours} | {minutes, Mins} | {seconds, Secs} | Millisecs | infinity
Hours = integer()
Mins = integer()
Secs = integer()
Millisecs = integer() | float()

This function, similar to `timer:sleep/1` in STDLIB, suspends the test case for a specified time. However, this function also multiplies `Time` with the `multiply_timetraps` value (if set) and under certain circumstances also scales up the time automatically if `scale_timetraps` is set to `true` (default is `false`).

`start_interactive() -> ok`

Starts `Common Test` in interactive mode.

From this mode, all test case support functions can be executed directly from the Erlang shell. The interactive mode can also be started from the OS command line with `ct_run -shell [-config File...]`.

If any functions (for example, Telnet or FTP) using "required configuration data" are to be called from the Erlang shell, configuration data must first be required with *`ct:require/2`*.

Example:

```
> ct:require(unix_telnet, unix).
ok
> ct_telnet:open(unix_telnet).
{ok,<0.105.0>}
> ct_telnet:cmd(unix_telnet, "ls .").
{ok,["ls","file1  ...",...]}
```

`step(TestDir, Suite, Case) -> Result`

Types:

`Case = atom()`

Steps through a test case with the debugger.

See also *`ct:run/3`*.

`step(TestDir, Suite, Case, Opts) -> Result`

Types:

`Case = atom()`
`Opts = [Opt] | []`
`Opt = config | keep_inactive`

Steps through a test case with the debugger. If option `config` has been specifed, breakpoints are also set on the configuration functions in `Suite`.

See also *`ct:run/3`*.

`stop_interactive() -> ok`

Exits the interactive mode.

See also *`ct:start_interactive/0`*.

`sync_notify(Name, Data) -> ok`

Types:

`Name = atom()`
`Data = term()`

Sends a synchronous notification of type `Name` with `Data` to the `Common Test` event manager. This can later be caught by any installed event manager.

See also *`gen_event(3)`*.

`testcases(TestDir, Suite) -> Testcases | {error, Reason}`

Types:

TestDir = string()
Suite = atom()
Testcases = list()
Reason = term()

Returns all test cases in the specified suite.

`timetrap(Time) -> ok`

Types:

Time = {hours, Hours} | {minutes, Mins} | {seconds, Secs} | Millisecs | infinity | Func
Hours = integer()
Mins = integer()
Secs = integer()
Millisecs = integer()
Func = {M, F, A} | function()
M = atom()
F = atom()
A = list()

Sets a new timetrap for the running test case.

If the argument is `Func`, the timetrap is triggered when this function returns. `Func` can also return a new `Time` value, which in that case is the value for the new timetrap.

`userdata(TestDir, Suite) -> SuiteUserData | {error, Reason}`

Types:

TestDir = string()
Suite = atom()
SuiteUserData = [term()]
Reason = term()

Returns any data specified with tag `userdata` in the list of tuples returned from *`suite/0`*.

`userdata(TestDir, Suite, Case::GroupOrCase) -> TCUserData | {error, Reason}`

Types:

TestDir = string()
Suite = atom()
GroupOrCase = {group, GroupName} | atom()
GroupName = atom()
TCUserData = [term()]
Reason = term()

Returns any data specified with tag `userdata` in the list of tuples returned from `Suite:group(GroupName)` or `Suite:Case()`.

ct_master

Erlang module

Distributed test execution control for `Common Test`.

This module exports functions for running `Common Test` nodes on multiple hosts in parallel.

Exports

`abort() -> ok`

Stops all running tests.

`abort(Nodes) -> ok`

Types:

`Nodes = atom() | [atom()]`

Stops tests on specified nodes.

`basic_html(Bool) -> ok`

Types:

`Bool = true | false`

If set to `true`, the `ct_master logs` are written on a primitive HTML format, not using the `Common Test` CSS style sheet.

`get_event_mgr_ref() -> MasterEvMgrRef`

Types:

`MasterEvMgrRef = atom()`

Gets a reference to the `Common Test` master event manager. The reference can be used to, for example, add a user-specific event handler while tests are running.

Example:

```
gen_event:add_handler(ct_master:get_event_mgr_ref(), my_ev_h, [])
```

`progress() -> [{Node, Status}]`

Types:

`Node = atom()`
`Status = finished_ok | ongoing | aborted | {error, Reason}`
`Reason = term()`

Returns test progress. If `Status` is `ongoing`, tests are running on the node and are not yet finished.

`run(TestSpecs) -> ok`

Types:

`TestSpecs = string() | [SeparateOrMerged]`

Equivalent to *`ct_master:run(TestSpecs, false, [], [])`*.

`run(TestSpecs, InclNodes, ExclNodes) -> ok`

Types:

```
TestSpecs = string() | [SeparateOrMerged]
SeparateOrMerged = string() | [string()]
InclNodes = [atom()]
ExclNodes = [atom()]
```

Equivalent to *`ct_master:run(TestSpecs, false, InclNodes, ExclNodes)`*.

`run(TestSpecs, AllowUserTerms, InclNodes, ExclNodes) -> ok`

Types:

```
TestSpecs = string() | [SeparateOrMerged]
SeparateOrMerged = string() | [string()]
AllowUserTerms = bool()
InclNodes = [atom()]
ExclNodes = [atom()]
```

Tests are spawned on the nodes as specified in `TestSpecs`. Each specification in `TestSpec` is handled separately. However, it is also possible to specify a list of specifications to be merged into one specification before the tests are executed. Any test without a particular node specification is also executed on the nodes in `InclNodes`. Nodes in the `ExclNodes` list are excluded from the test.

`run_on_node(TestSpecs, Node) -> ok`

Types:

```
TestSpecs = string() | [SeparateOrMerged]
SeparateOrMerged = string() | [string()]
Node = atom()
```

Equivalent to *`ct_master:run_on_node(TestSpecs, false, Node)`*.

`run_on_node(TestSpecs, AllowUserTerms, Node) -> ok`

Types:

```
TestSpecs = string() | [SeparateOrMerged]
SeparateOrMerged = string() | [string()]
AllowUserTerms = bool()
Node = atom()
```

Tests are spawned on `Node` according to `TestSpecs`.

`run_test(Node, Opts) -> ok`

Types:

```
Node = atom()
Opts = [OptTuples]
OptTuples = {config, CfgFiles} | {dir, TestDirs} | {suite, Suites}
| {testcase, Cases} | {spec, TestSpecs} | {allow_user_terms, Bool} |
{logdir, LogDir} | {event_handler, EventHandlers} | {silent_connections,
Conns} | {cover, CoverSpecFile} | {cover_stop, Bool} | {userconfig,
UserCfgFiles}
```

```
CfgFiles = string() | [string()]
TestDirs = string() | [string()]
Suites = atom() | [atom()]
Cases = atom() | [atom()]
TestSpecs = string() | [string()]
LogDir = string()
EventHandlers = EH | [EH]
EH = atom() | {atom(), InitArgs} | {[atom()], InitArgs}
InitArgs = [term()]
Conns = all | [atom()]
```

Tests are spawned on Node using *ct:run_test/1*

ct_cover

Erlang module

`Common Test` framework code coverage support module.

This module exports help functions for performing code coverage analysis.

Exports

`add_nodes(Nodes) -> {ok, StartedNodes} | {error, Reason}`

Types:

`Nodes = [atom()]`
`StartedNodes = [atom()]`
`Reason = cover_not_running | not_main_node`

Adds nodes to current cover test. Notice that this only works if cover support is active.

To have effect, this function is to be called from `init_per_suite/1` (see *`common_test`*) before any tests are performed.

`cross_cover_analyse(Level, Tests) -> ok`

Types:

`Level = overview | details`
`Tests = [{Tag, Dir}]`
`Tag = atom()`
`Dir = string()`

Accumulates cover results over multiple tests. See section *Cross Cover Analysis* in the Users's Guide.

`remove_nodes(Nodes) -> ok | {error, Reason}`

Types:

`Nodes = [atom()]`
`Reason = cover_not_running | not_main_node`

Removes nodes from the current cover test.

Call this function to stop cover test on nodes previously added with *`ct_cover:add_nodes/1`*. Results on the remote node are transferred to the `Common Test` node.

ct_ftp

Erlang module

FTP client module (based on the `ftp` application).

Data Types

`connection() = handle() | target_name()`

For `target_name`, see module *`ct`*.

`handle() = handle()`

Handle for a specific FTP connection, see module *`ct`*.

Exports

`cd(Connection, Dir) -> ok | {error, Reason}`

Types:

`Connection = connection()`
`Dir = string()`

Changes directory on remote host.

`close(Connection) -> ok | {error, Reason}`

Types:

`Connection = connection()`

Closes the FTP connection.

`delete(Connection, File) -> ok | {error, Reason}`

Types:

`Connection = connection()`
`File = string()`

Deletes a file on remote host.

`get(KeyOrName, RemoteFile, LocalFile) -> ok | {error, Reason}`

Types:

`KeyOrName = Key | Name`
`Key = atom()`
`Name = target_name()`
`RemoteFile = string()`
`LocalFile = string()`

Opens an FTP connection and fetches a file from the remote host.

`RemoteFile` and `LocalFile` must be absolute paths.

The configuration file must be as for *`ct_ftp:put/3`*.

For `target_name`, see module *`ct`*.

See also *ct:require/2*.

`ls(Connection, Dir) -> {ok, Listing} | {error, Reason}`

Types:

Connection = connection()
Dir = string()
Listing = string()

Lists directory `Dir`.

`open(KeyOrName) -> {ok, Handle} | {error, Reason}`

Types:

KeyOrName = Key | Name
Key = atom()
Name = target_name()
Handle = handle()

Opens an FTP connection to the specified node.

You can open a connection for a particular `Name` and use the same name as reference for all following subsequent operations. If you want the connection to be associated with `Handle` instead (if you, for example, need to open multiple connections to a host), use `Key`, the configuration variable name, to specify the target. A connection without an associated target name can only be closed with the handle value.

For information on how to create a new `Name`, see *ct:require/2*.

For `target_name`, see module *ct*.

`put(KeyOrName, LocalFile, RemoteFile) -> ok | {error, Reason}`

Types:

KeyOrName = Key | Name
Key = atom()
Name = target_name()
LocalFile = string()
RemoteFile = string()

Opens an FTP connection and sends a file to the remote host.

`LocalFile` and `RemoteFile` must be absolute paths.

For `target_name`, see module *ct*.

If the target host is a "special" node, the FTP address must be specified in the configuration file as follows:

```
{node,[{ftp,IpAddr}]}.
```

If the target host is something else, for example, a UNIX host, the configuration file must also include the username and password (both strings):

```
{unix,[{ftp,IpAddr},
       {username,Username},
       {password,Password}]}.
```

See also *ct:require/2*.

`recv(Connection, RemoteFile) -> ok | {error, Reason}`

Fetches a file over FTP.

The file gets the same name on the local host.

See also *ct_ftp:recv/3*.

`recv(Connection, RemoteFile, LocalFile) -> ok | {error, Reason}`

Types:

`Connection = connection()`
`RemoteFile = string()`
`LocalFile = string()`

Fetches a file over FTP.

The file is named `LocalFile` on the local host.

`send(Connection, LocalFile) -> ok | {error, Reason}`

Sends a file over FTP.

The file gets the same name on the remote host.

See also *ct_ftp:send/3*.

`send(Connection, LocalFile, RemoteFile) -> ok | {error, Reason}`

Types:

`Connection = connection()`
`LocalFile = string()`
`RemoteFile = string()`

Sends a file over FTP.

The file is named `RemoteFile` on the remote host.

`type(Connection, Type) -> ok | {error, Reason}`

Types:

`Connection = connection()`
`Type = ascii | binary`

Changes the file transfer type.

ct_ssh

Erlang module

SSH/SFTP client module.

This module uses application SSH, which provides detailed information about, for example, functions, types, and options.

Argument Server in the SFTP functions is only to be used for SFTP sessions that have been started on existing SSH connections (that is, when the original connection type is ssh). Whenever the connection type is sftp, use the SSH connection reference only.

The following options are valid for specifying an SSH/SFTP connection (that is, can be used as configuration elements):

```
[{ConnType, Addr},
 {port, Port},
 {user, UserName}
 {password, Pwd}
 {user_dir, String}
 {public_key_alg, PubKeyAlg}
 {connect_timeout, Timeout}
 {key_cb, KeyCallbackMod}]
```

ConnType = ssh | sftp.

For other types, see *ssh(3)*.

All time-out parameters in ct_ssh functions are values in milliseconds.

Data Types

connection() = handle() | target_name()

For target_name, see module *ct*.

handle() = handle()

Handle for a specific SSH/SFTP connection, see module *ct*.

ssh_sftp_return() = term()

Return value from an *ssh_sftp* function.

Exports

apread(SSH, Handle, Position, Length) -> Result

Types:

SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()

For information and other types, see *ssh_sftp(3)*.

apread(SSH, Server, Handle, Position, Length) -> Result

Types:

SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()

For information and other types, see *ssh_sftp(3)*.

`apwrite(SSH, Handle, Position, Data) -> Result`

Types:

SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()

For information and other types, see *ssh_sftp(3)*.

`apwrite(SSH, Server, Handle, Position, Data) -> Result`

Types:

SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()

For information and other types, see *ssh_sftp(3)*.

`aread(SSH, Handle, Len) -> Result`

Types:

SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()

For information and other types, see *ssh_sftp(3)*.

`aread(SSH, Server, Handle, Len) -> Result`

Types:

SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()

For information and other types, see *ssh_sftp(3)*.

`awrite(SSH, Handle, Data) -> Result`

Types:

SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()

For information and other types, see *ssh_sftp(3)*.

`awrite(SSH, Server, Handle, Data) -> Result`

Types:

SSH = connection()

Result = ssh_sftp_return() | {error, Reason}
Reason = term()

For information and other types, see *ssh_sftp(3)*.

`close(SSH, Handle) -> Result`

Types:

SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()

For information and other types, see *ssh_sftp(3)*.

`close(SSH, Server, Handle) -> Result`

Types:

SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()

For information and other types, see *ssh_sftp(3)*.

`connect(KeyOrName) -> {ok, Handle} | {error, Reason}`

Equivalent to *ct_ssh:connect(KeyOrName, host, [])*.

`connect(KeyOrName, ConnType) -> {ok, Handle} | {error, Reason}`

Equivalent to *ct_ssh:connect(KeyOrName, ConnType, [])*.

`connect(KeyOrName, ConnType, ExtraOpts) -> {ok, Handle} | {error, Reason}`

Types:

KeyOrName = Key | Name
Key = atom()
Name = target_name()
ConnType = ssh | sftp | host
ExtraOpts = ssh_connect_options()
Handle = handle()
Reason = term()

Opens an SSH or SFTP connection using the information associated with `KeyOrName`.

If `Name` (an alias name for `Key`) is used to identify the connection, this name can be used as connection reference for subsequent calls. Only one open connection at a time associated with `Name` is possible. If `Key` is used, the returned handle must be used for subsequent calls (multiple connections can be opened using the configuration data specified by `Key`).

For information on how to create a new `Name`, see *ct:require/2*.

For `target_name`, see module *ct*.

`ConnType` always overrides the type specified in the address tuple in the configuration data (and in `ExtraOpts`). So it is possible to, for example, open an SFTP connection directly using data originally specifying an SSH connection.

Value `host` means that the connection type specified by the host option (either in the configuration data or in `ExtraOpts`) is used.

`ExtraOpts` (optional) are extra SSH options to be added to the configuration data for `KeyOrName`. The extra options override any existing options with the same key in the configuration data. For details on valid SSH options, see application *SSH*.

`del_dir(SSH, Name) -> Result`

Types:

SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()

For information and other types, see *ssh_sftp(3)*.

`del_dir(SSH, Server, Name) -> Result`

Types:

SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()

For information and other types, see *ssh_sftp(3)*.

`delete(SSH, Name) -> Result`

Types:

SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()

For information and other types, see *ssh_sftp(3)*.

`delete(SSH, Server, Name) -> Result`

Types:

SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()

For information and other types, see *ssh_sftp(3)*.

`disconnect(SSH) -> ok | {error, Reason}`

Types:

SSH = connection()
Reason = term()

Closes an SSH/SFTP connection.

`exec(SSH, Command) -> {ok, Data} | {error, Reason}`

Equivalent to *ct_ssh:exec(SSH, Command, DefaultTimeout)*.

`exec(SSH, Command, Timeout) -> {ok, Data} | {error, Reason}`

Types:

`SSH = connection()`
`Command = string()`
`Timeout = integer()`
`Data = list()`
`Reason = term()`

Requests server to perform `Command`. A session channel is opened automatically for the request. `Data` is received from the server as a result of the command.

`exec(SSH, ChannelId, Command, Timeout) -> {ok, Data} | {error, Reason}`

Types:

`SSH = connection()`
`ChannelId = integer()`
`Command = string()`
`Timeout = integer()`
`Data = list()`
`Reason = term()`

Requests server to perform `Command`. A previously opened session channel is used for the request. `Data` is received from the server as a result of the command.

`get_file_info(SSH, Handle) -> Result`

Types:

`SSH = connection()`
`Result = ssh_sftp_return() | {error, Reason}`
`Reason = term()`

For information and other types, see *`ssh_sftp(3)`*.

`get_file_info(SSH, Server, Handle) -> Result`

Types:

`SSH = connection()`
`Result = ssh_sftp_return() | {error, Reason}`
`Reason = term()`

For information and other types, see *`ssh_sftp(3)`*.

`list_dir(SSH, Path) -> Result`

Types:

`SSH = connection()`
`Result = ssh_sftp_return() | {error, Reason}`
`Reason = term()`

For information and other types, see *`ssh_sftp(3)`*.

`list_dir(SSH, Server, Path) -> Result`

Types:

```
SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()
```

For information and other types, see *ssh_sftp(3)*.

`make_dir(SSH, Name) -> Result`

Types:

```
SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()
```

For information and other types, see *ssh_sftp(3)*.

`make_dir(SSH, Server, Name) -> Result`

Types:

```
SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()
```

For information and other types, see *ssh_sftp(3)*.

`make_symlink(SSH, Name, Target) -> Result`

Types:

```
SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()
```

For information and other types, see *ssh_sftp(3)*.

`make_symlink(SSH, Server, Name, Target) -> Result`

Types:

```
SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()
```

For information and other types, see *ssh_sftp(3)*.

`open(SSH, File, Mode) -> Result`

Types:

```
SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()
```

For information and other types, see *ssh_sftp(3)*.

`open(SSH, Server, File, Mode) -> Result`

Types:

SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()

For information and other types, see *ssh_sftp(3)*.

`opendir(SSH, Path) -> Result`

Types:

SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()

For information and other types, see *ssh_sftp(3)*.

`opendir(SSH, Server, Path) -> Result`

Types:

SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()

For information and other types, see *ssh_sftp(3)*.

`position(SSH, Handle, Location) -> Result`

Types:

SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()

For information and other types, see *ssh_sftp(3)*.

`position(SSH, Server, Handle, Location) -> Result`

Types:

SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()

For information and other types, see *ssh_sftp(3)*.

`pread(SSH, Handle, Position, Length) -> Result`

Types:

SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()

For information and other types, see *ssh_sftp(3)*.

```
pread(SSH, Server, Handle, Position, Length) -> Result
```

Types:

```
SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()
```

For information and other types, see *ssh_sftp(3)*.

```
pwrite(SSH, Handle, Position, Data) -> Result
```

Types:

```
SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()
```

For information and other types, see *ssh_sftp(3)*.

```
pwrite(SSH, Server, Handle, Position, Data) -> Result
```

Types:

```
SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()
```

For information and other types, see *ssh_sftp(3)*.

```
read(SSH, Handle, Len) -> Result
```

Types:

```
SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()
```

For information and other types, see *ssh_sftp(3)*.

```
read(SSH, Server, Handle, Len) -> Result
```

Types:

```
SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()
```

For information and other types, see *ssh_sftp(3)*.

```
read_file(SSH, File) -> Result
```

Types:

```
SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()
```

For information and other types, see *ssh_sftp(3)*.

`read_file(SSH, Server, File) -> Result`

Types:

`SSH = connection()`
`Result = ssh_sftp_return() | {error, Reason}`
`Reason = term()`

For information and other types, see *`ssh_sftp(3)`*.

`read_file_info(SSH, Name) -> Result`

Types:

`SSH = connection()`
`Result = ssh_sftp_return() | {error, Reason}`
`Reason = term()`

For information and other types, see *`ssh_sftp(3)`*.

`read_file_info(SSH, Server, Name) -> Result`

Types:

`SSH = connection()`
`Result = ssh_sftp_return() | {error, Reason}`
`Reason = term()`

For information and other types, see *`ssh_sftp(3)`*.

`read_link(SSH, Name) -> Result`

Types:

`SSH = connection()`
`Result = ssh_sftp_return() | {error, Reason}`
`Reason = term()`

For information and other types, see *`ssh_sftp(3)`*.

`read_link(SSH, Server, Name) -> Result`

Types:

`SSH = connection()`
`Result = ssh_sftp_return() | {error, Reason}`
`Reason = term()`

For information and other types, see *`ssh_sftp(3)`*.

`read_link_info(SSH, Name) -> Result`

Types:

`SSH = connection()`
`Result = ssh_sftp_return() | {error, Reason}`
`Reason = term()`

For information and other types, see *`ssh_sftp(3)`*.

`read_link_info(SSH, Server, Name) -> Result`

Types:

`SSH = connection()`
`Result = ssh_sftp_return() | {error, Reason}`
`Reason = term()`

For information and other types, see *`ssh_sftp(3)`*.

`receive_response(SSH, ChannelId) -> {ok, Data} | {error, Reason}`

Equivalent to *`ct_ssh:receive_response(SSH, ChannelId, close)`*.

`receive_response(SSH, ChannelId, End) -> {ok, Data} | {error, Reason}`

Equivalent to *`ct_ssh:receive_response(SSH, ChannelId, End, DefaultTimeout)`*.

`receive_response(SSH, ChannelId, End, Timeout) -> {ok, Data} | {timeout, Data} | {error, Reason}`

Types:

`SSH = connection()`
`ChannelId = integer()`
`End = Fun | close | timeout`
`Timeout = integer()`
`Data = list()`
`Reason = term()`

Receives expected data from server on the specified session channel.

If `End == close`, data is returned to the caller when the channel is closed by the server. If a time-out occurs before this happens, the function returns `{timeout,Data}` (where `Data` is the data received so far).

If `End == timeout`, a time-out is expected and `{ok,Data}` is returned both in the case of a time-out and when the channel is closed.

If `End` is a fun, this fun is called with one argument, the data value in a received `ssh_cm` message (see *`ssh_connection(3)`*. The fun is to return either `true` to end the receiving operation (and have the so far collected data returned) or `false` to wait for more data from the server. Even if a fun is supplied, the function returns immediately if the server closes the channel).

`rename(SSH, OldName, NewName) -> Result`

Types:

`SSH = connection()`
`Result = ssh_sftp_return() | {error, Reason}`
`Reason = term()`

For information and other types, see *`ssh_sftp(3)`*.

`rename(SSH, Server, OldName, NewName) -> Result`

Types:

`SSH = connection()`
`Result = ssh_sftp_return() | {error, Reason}`

Reason = term()

For information and other types, see *ssh_sftp(3)*.

`send(SSH, ChannelId, Data) -> ok | {error, Reason}`

Equivalent to *ct_ssh:send(SSH, ChannelId, 0, Data, DefaultTimeout)*.

`send(SSH, ChannelId, Data, Timeout) -> ok | {error, Reason}`

Equivalent to *ct_ssh:send(SSH, ChannelId, 0, Data, Timeout)*.

`send(SSH, ChannelId, Type, Data, Timeout) -> ok | {error, Reason}`

Types:

SSH = connection()
ChannelId = integer()
Type = integer()
Data = list()
Timeout = integer()
Reason = term()

Sends data to server on specified session channel.

`send_and_receive(SSH, ChannelId, Data) -> {ok, Data} | {error, Reason}`

Equivalent to *ct_ssh:send_and_receive(SSH, ChannelId, Data, close)*.

`send_and_receive(SSH, ChannelId, Data, End) -> {ok, Data} | {error, Reason}`

Equivalent to *ct_ssh;send_and_receive(SSH, ChannelId, 0, Data, End, DefaultTimeout)*.

`send_and_receive(SSH, ChannelId, Data, End, Timeout) -> {ok, Data} | {error, Reason}`

Equivalent to *ct_ssh:send_and_receive(SSH, ChannelId, 0, Data, End, Timeout)*.

`send_and_receive(SSH, ChannelId, Type, Data, End, Timeout) -> {ok, Data} | {error, Reason}`

Types:

SSH = connection()
ChannelId = integer()
Type = integer()
Data = list()
End = Fun | close | timeout
Timeout = integer()
Reason = term()

Sends data to server on specified session channel and waits to receive the server response.

For details on argument End, see *ct_ssh:receive_response/4*.

```
session_close(SSH, ChannelId) -> ok | {error, Reason}
```

Types:

SSH = connection()
ChannelId = integer()
Reason = term()

Closes an SSH session channel.

```
session_open(SSH) -> {ok, ChannelId} | {error, Reason}
```

Equivalent to *ct_ssh:session_open(SSH, DefaultTimeout)*.

```
session_open(SSH, Timeout) -> {ok, ChannelId} | {error, Reason}
```

Types:

SSH = connection()
Timeout = integer()
ChannelId = integer()
Reason = term()

Opens a channel for an SSH session.

```
sftp_connect(SSH) -> {ok, Server} | {error, Reason}
```

Types:

SSH = connection()
Server = pid()
Reason = term()

Starts an SFTP session on an already existing SSH connection. `Server` identifies the new session and must be specified whenever SFTP requests are to be sent.

```
shell(SSH, ChannelId) -> ok | {error, Reason}
```

Equivalent to *ct_ssh:shell(SSH, ChannelId, DefaultTimeout)*.

```
shell(SSH, ChannelId, Timeout) -> ok | {error, Reason}
```

Types:

SSH = connection()
ChannelId = integer()
Timeout = integer()
Reason = term()

Requests that the user default shell (typically defined in `/etc/passwd` in Unix systems) is executed at the server end.

```
subsystem(SSH, ChannelId, Subsystem) -> Status | {error, Reason}
```

Equivalent to *ct_ssh:subsystem(SSH, ChannelId, Subsystem, DefaultTimeout)*.

```
subsystem(SSH, ChannelId, Subsystem, Timeout) -> Status | {error, Reason}
```

Types:

SSH = connection()

ChannelId = integer()
Subsystem = string()
Timeout = integer()
Status = success | failure
Reason = term()

Sends a request to execute a predefined subsystem.

`write(SSH, Handle, Data) -> Result`

Types:

SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()

For information and other types, see *ssh_sftp(3)*.

`write(SSH, Server, Handle, Data) -> Result`

Types:

SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()

For information and other types, see *ssh_sftp(3)*.

`write_file(SSH, File, Iolist) -> Result`

Types:

SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()

For information and other types, see *ssh_sftp(3)*.

`write_file(SSH, Server, File, Iolist) -> Result`

Types:

SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()

For information and other types, see *ssh_sftp(3)*.

`write_file_info(SSH, Name, Info) -> Result`

Types:

SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()

For information and other types, see *ssh_sftp(3)*.

`write_file_info(SSH, Server, Name, Info) -> Result`

Types:

`SSH = connection()`

`Result = ssh_sftp_return() | {error, Reason}`

`Reason = term()`

For information and other types, see *`ssh_sftp(3)`*.

ct_netconfc

Erlang module

NETCONF client module compliant with RFC 6241, NETCONF Configuration Protocol, and RFC 6242, Using the NETCONF Configuration Protocol over Secure SHell (SSH), and with support for RFC 5277, NETCONF Event Notifications.

Connecting to a NETCONF server

Call *`connect/1,2`* to establish a connection to a server, then pass the returned handle to *`session/1-3`* to establish a NETCONF session on a new SSH channel. Each call to *`session/1-3`* establishes a new session on the same connection, and results in a hello message to the server.

Alternately, *`open/1,2`* can be used to establish a single session on a dedicated connection. (Or, equivalently, *`only_open/1,2`* followed by *`hello/1-3`*.)

Connect/session options can be specified in a configuration file with entries like the following.

```
{server_id(), [option()]}.
```

The *`server_id()`* or an associated *`ct:target_name()`* can then be passed to the aforementioned functions to use the referenced configuration.

Signaling

Protocol operations in the NETCONF protocol are realized as remote procedure calls (RPCs) from client to server and a corresponding reply from server to client. RPCs are sent using like-named functions (eg. *`edit_config/3-5`* to send an edit-config RPC), with the server reply as return value. There are functions for each RPC defined in RFC 6241 and the create-subscription RPC from RFC 5277, all of which are wrappers on *`send_rpc/2,3`*, that can be used to send an arbitrary RPC not defined in RFC 6241 or RFC 5277.

All of the signaling functions have one variant with a `Timeout` argument and one without, corresponding to an infinite timeout. The latter is inappropriate in most cases since a non-response by the server or a missing message-id causes the call to hang indefinitely.

Logging

The NETCONF server uses `error_logger` for logging of NETCONF traffic. A special purpose error handler is implemented in `ct_conn_log_h`. To use this error handler, add the `cth_conn_log` hook in the test suite, for example:

```
suite() ->
   [{ct_hooks, [{cth_conn_log, [{ct:conn_log_mod(), ct:conn_log_options()}]}]}].
```

`conn_log_mod()` is the name of the `Common Test` module implementing the connection protocol, for example, `ct_netconfc`.

Hook option `log_type` specifies the type of logging:

`raw`

The sent and received NETCONF data is logged to a separate text file "as is" without any formatting. A link to the file is added to the test case HTML log.

`pretty`

The sent and received NETCONF data is logged to a separate text file with XML data nicely indented. A link to the file is added to the test case HTML log.

`html (default)`

The sent and received NETCONF traffic is pretty printed directly in the test case HTML log.

`silent`

NETCONF traffic is not logged.

By default, all NETCONF traffic is logged in one single log file. However, different connections can be logged in separate files. To do this, use hook option `hosts` and list the names of the servers/connections to be used in the suite. The connections must be named for this to work, that is, they must be opened with *`open/2`*.

Option `hosts` has no effect if `log_type` is set to `html` or `silent`.

The hook options can also be specified in a configuration file with configuration variable `ct_conn_log`:

```
{ct_conn_log,[{ct:conn_log_mod(), ct:conn_log_options()}]}.
```

For example:

```
{ct_conn_log,[{ct_netconfc,[{log_type,pretty},
                            {hosts,[ct:key_or_name()]}]}]}
```

Note:

Hook options specified in a configuration file overwrite the hard-coded hook options in the test suite.

Logging Example 1:

The following `ct_hooks` statement causes pretty printing of NETCONF traffic to separate logs for the connections named `nc_server1` and `nc_server2`. Any other connections are logged to default NETCONF log.

```
suite() ->
   [{ct_hooks, [{cth_conn_log, [{ct_netconfc,[{log_type,pretty}},
                                              {hosts,[nc_server1,nc_server2]}]}
                               ]}]}].
```

Connections must be opened as follows:

```
open(nc_server1,[...]),
open(nc_server2,[...]).
```

Logging Example 2:

The following configuration file causes raw logging of all NETCONF traffic in to one single text file:

```
{ct_conn_log,[{ct_netconfc,[{log_type,raw}]}]}.
```

The `ct_hooks` statement must look as follows:

```
suite() ->
   [{ct_hooks, [{cth_conn_log, []}]}].
```

The same `ct_hooks` statement without the configuration file would cause HTML logging of all NETCONF connections in to the test case HTML log.

Data Types

`client() = handle() | server_id() | ct:target_name()`

Handle to a NETCONF session, as required by signaling functions.

`handle()`

Handle to a connection to a NETCONF server as returned by *`connect/1,2`*, or to a session as returned by *`session/1-3`*, *`open/1,2`*, or *`only_open/1,2`*.

`xs_datetime() = string()`

Date and time of a startTime/stopTime element in an RFC 5277 create-subscription request. Of XML primitive type `dateTime`, which has the (informal) form

```
[-]YYYY-MM-DDThh:mm:ss[.s][Z|(+|-)hh:mm]
```

where `T` and `Z` are literal and `.s` is one or more fractional seconds.

```
event_time() = {eventTime, xml_attributes(), [xs_datetime()]}
notification_content() = [event_time() | simple_xml()]
notification() =
    {notification, xml_attributes(), notification_content()}
```

Event notification messages sent as a result of calls to *`create_subscription/2,3`*.

```
option() =
    {host | ssh, host()} |
    {port, inet:port_number()} |
    {timeout, timeout()} |
    {capability, string() | [string()]} |
    ssh:client_option()
```

Options `host` and `port` specify the server endpoint to which to connect, and are passed directly to *`ssh:connect/4`*, as are arbitrary ssh options. Common options are `user`, `password` and `user_dir`.

Option `timeout` specifies the number of milliseconds to allow for connection establishment and, if the function in question results in an outgoing hello message, reception of the server hello. The timeout applies to connection and hello independently; one timeout for connection establishment, another for hello reception.

Option `capability` specifies the content of a corresponding element in an outgoing hello message, each option specifying the content of a single element. If no base NETCONF capability is configured then the RFC 4741 1.0 capability, "urn:ietf:params:netconf:base:1.0", is added, otherwise not. In particular, the RFC 6241 1.1 capability must be explicitly configured. NETCONF capabilities can be specified using the shorthand notation defined in RFC 6241, any capability string starting with a colon being prefixed by either "urn:ietf:params:netconf" or "urn:ietf:params:netconf:capability", as appropriate.

Capability options are ignored by connect/1-3 and only_open/1-2, which don't result in an outgoing hello message.

`server_id() = atom()`

Identity of connection or session configuration in a configuration file.

```
stream_data() =
    {description, string()} |
    {replaySupport, string()} |
    {replayLogCreationTime, string()} |
```

```
    {replayLogAgedTime, string()}
stream_name() = string()
streams() = [{stream_name(), [stream_data()]}]
```

Stream information as returned by *get_event_streams/1-3*. See RFC 5277, "XML Schema for Event Notifications", for detail on the format of the string values.

```
xml_attribute_tag() = atom()
xml_attribute_value() = string()
xml_attributes() =
    [{xml_attribute_tag(), xml_attribute_value()}]
xml_content() = [simple_xml() | iolist()]
xml_tag() = atom()
simple_xml() =
    {xml_tag(), xml_attributes(), xml_content()} |
    {xml_tag(), xml_content()} |
    xml_tag()
```

Representation of XML, as described in application *xmerl*.

```
xpath() = {xpath, string()}
error_reason() = term()
host() = inet:hostname() | inet:ip_address()
netconf_db() = running | startup | candidate
```

Exports

```
action(Client, Action) -> Result
action(Client, Action, Timeout) -> Result
```

Types:

```
   Client = client()
   Action = simple_xml()
   Timeout = timeout()
   Result = ok | {ok, [simple_xml()]} | {error, error_reason()}
```

Executes an action. If the return type is void, `ok` is returned instead of `{ok,[simple_xml()]}`.

```
close_session(Client) -> Result
close_session(Client, Timeout) -> Result
```

Types:

```
   Client = client()
   Timeout = timeout()
   Result = ok | {error, error_reason()}
```

Requests graceful termination of the session associated with the client.

When a NETCONF server receives a `close-session` request, it gracefully closes the session. The server releases any locks and resources associated with the session and gracefully closes any associated connections. Any NETCONF requests received after a `close-session` request are ignored.

```
connect(Options) -> Result
```

Types:

```
Options = [option()]
Result = {ok, handle()} | {error, error_reason()}
```

Opens an SSH connection to a NETCONF server.

If the server options are specified in a configuration file, use *connect/2* instead.

The opaque *handle()* reference returned from this function is required as connection identifier when opening sessions over this connection, see *session/1-3*.

```
connect(KeyOrName, ExtraOptions) -> Result
```

Types:

```
KeyOrName = ct:key_or_name()
ExtraOptions = [option()]
Result = {ok, handle()} | {error, error_reason()}
```

Open an SSH connection to a named NETCONF server.

If `KeyOrName` is a configured `server_id()` or a `target_name()` associated with such an Id, then the options for this server are fetched from the configuration file.

The options list is added to those of the configuration file. If an option is specified in both lists, the configuration file takes precedence.

If the server is not specified in a configuration file, use *connect/1* instead.

The opaque *handle()* reference returned from this function can be used as connection identifier when opening sessions over this connection, see *session/1-3*. However, if `KeyOrName` is a `target_name()`, that is, if the server is named through a call to *ct:require/2* or a `require` statement in the test suite, then this name can be used instead of *handle()*.

```
copy_config(Client, Target, Source) -> Result
copy_config(Client, Target, Source, Timeout) -> Result
```

Types:

```
Client = client()
Target = Source = netconf_db()
Timeout = timeout()
Result = ok | {error, error_reason()}
```

Copies configuration data.

Which source and target options that can be issued depends on the capabilities supported by the server. That is, `:candidate` and/or `:startup` are required.

```
create_subscription(Client, Values) -> Result
create_subscription(Client, Values, Timeout) -> Result
```

Types:

```
Client = client()
Values =
    #{stream => Stream,
      filter => Filter,
      start => StartTime,
      stop => StopTime}
Stream = stream_name()
Filter = simple_xml() | [simple_xml()]
StartTime = StopTime = xs_datetime()
Timeout = timeout()
Result = ok | {error, error_reason()}
```

Creates a subscription for event notifications by sending an RFC 5277 create-subscription RPC to the server. The calling process receives events as messages of type `notification()`.

From RFC 5722, 2.1 Subscribing to Receive Event Notifications:

`Stream`

Indicates which stream of event is of interest. If not present, events in the default NETCONF stream are sent.

`Filter`

Indicates which subset of all possible events is of interest. The parameter format is the same as that of the filter parameter in the NETCONF protocol operations. If not present, all events not precluded by other parameters are sent.

`StartTime`

Used to trigger the replay feature and indicate that the replay is to start at the time specified. If `StartTime` is not present, this is not a replay subscription. It is not valid to specify start times that are later than the current time. If `StartTime` is specified earlier than the log can support, the replay begins with the earliest available notification. This parameter is of type `dateTime` and compliant to RFC 3339. Implementations must support time zones.

`StopTime`

Used with the optional replay feature to indicate the newest notifications of interest. If `StopTime` is not present, the notifications continues until the subscription is terminated. Must be used with and be later than `StartTime`. Values of `StopTime` in the future are valid. This parameter is of type `dateTime` and compliant to RFC 3339. Implementations must support time zones.

See RFC 5277 for more details. The requirement that `StopTime` must only be used with `StartTime` is not enforced, to allow an invalid request to be sent to the server.

Prior to OTP 22.1, this function was documented as having 15 variants in 6 arities. These are still exported for backwards compatibility, but no longer documented. The map-based variants documented above provide the same functionality with simpler arguments.

```
delete_config(Client, Target) -> Result
delete_config(Client, Target, Timeout) -> Result
```

Types:

```
Client = client()
Target = startup | candidate
Timeout = timeout()
Result = ok | {error, error_reason()}
```

Deletes configuration data.

The running configuration cannot be deleted and `:candidate` or `:startup` must be advertised by the server.

```
disconnect(Conn) -> ok | {error, error_reason()}
```

Types:

```
Conn = handle()
```

Closes the given SSH connection.

If there are open NETCONF sessions on the connection, these will be brutally aborted. To avoid this, close each session with *`close_session/1,2`*

```
edit_config(Client, Target, Config) -> Result
edit_config(Client, Target, Config, OptParams) -> Result
edit_config(Client, Target, Config, Timeout) -> Result
edit_config(Client, Target, Config, OptParams, Timeout) -> Result
```

Types:

```
Client = client()
Target = netconf_db()
Config = simple_xml() | [simple_xml()]
OptParams = [simple_xml()]
Timeout = timeout()
Result = ok | {error, error_reason()}
```

Edits configuration data.

By default only the running target is available, unless the server includes `:candidate` or `:startup` in its list of capabilities.

`OptParams` can be used for specifying optional parameters (`default-operation`, `test-option`, or `error-option`) to be added to the `edit-config` request. The value must be a list containing valid simple XML, for example:

```
[{'default-operation', ["none"]},
 {'error-option', ["rollback-on-error"]}]
```

If `OptParams` is not given, the default value `[]` is used.

```
get(Client, Filter) -> Result
get(Client, Filter, Timeout) -> Result
```

Types:

```
Client = client()
Filter = simple_xml() | xpath()
Timeout = timeout()
Result = {ok, [simple_xml()]} | {error, error_reason()}
```

Gets data.

This operation returns both configuration and state data from the server.

Filter type `xpath` can be used only if the server supports `:xpath`.

```
get_capabilities(Client) -> Result
get_capabilities(Client, Timeout) -> Result
```

Types:

```
Client = client()
Timeout = timeout()
Result = [string()] | {error, error_reason()}
```

Returns the server capabilities as received in its hello message.

```
get_config(Client, Source, Filter) -> Result
get_config(Client, Source, Filter, Timeout) -> Result
```

Types:

```
Client = client()
Source = netconf_db()
Filter = simple_xml() | xpath()
Timeout = timeout()
Result = {ok, [simple_xml()]} | {error, error_reason()}
```

Gets configuration data.

To be able to access another source than `running`, the server must advertise `:candidate` and/or `:startup`.

Filter type `xpath` can be used only if the server supports `:xpath`.

```
get_event_streams(Client) -> Result
get_event_streams(Client, Timeout) -> Result
get_event_streams(Client, Streams) -> Result
get_event_streams(Client, Streams, Timeout) -> Result
```

Types:

```
Client = client()
Streams = [stream_name()]
Timeout = timeout()
Result = {ok, streams()} | {error, error_reason()}
```

Sends a request to get the specified event streams.

`Streams` is a list of stream names. The following filter is sent to the NETCONF server in a `get` request:

```
<netconf xmlns="urn:ietf:params:xml:ns:netmod:notification">
  <streams>
    <stream>
      <name>StreamName1</name>
    </stream>
    <stream>
      <name>StreamName2</name>
    </stream>
    ...
  </streams>
</netconf>
```

If `Streams` is an empty list, **all** streams are requested by sending the following filter:

```
<netconf xmlns="urn:ietf:params:xml:ns:netmod:notification">
  <streams/>
</netconf>
```

If more complex filtering is needed, use *ct_netconfc:get/2,3* and specify the exact filter according to "XML Schema for Event Notifications" in RFC 5277.

```
get_session_id(Client) -> Result
get_session_id(Client, Timeout) -> Result
```

Types:

```
Client = client()
Timeout = timeout()
Result = integer() >= 1 | {error, error_reason()}
```

Returns the session Id associated with the specified client.

```
hello(Client) -> Result
hello(Client, Timeout) -> Result
hello(Client, Options, Timeout) -> Result
```

Types:

```
Client = handle()
Options = [{capability, [string()]}]
Timeout = timeout()
Result = ok | {error, error_reason()}
```

Exchanges `hello` messages with the server. Returns when the server hello has been received or after the specified timeout.

Note that capabilities for an outgoing hello can be passed directly to *open/2*.

```
kill_session(Client, SessionId) -> Result
kill_session(Client, SessionId, Timeout) -> Result
```

Types:

```
Client = client()
SessionId = integer() >= 1
Timeout = timeout()
Result = ok | {error, error_reason()}
```

Forces termination of the session associated with the supplied session Id.

The server side must abort any ongoing operations, release any locks and resources associated with the session, and close any associated connections.

Only if the server is in the confirmed commit phase, the configuration is restored to its state before entering the confirmed commit phase. Otherwise, no configuration rollback is performed.

If the specified `SessionId` is equal to the current session Id, an error is returned.

```
lock(Client, Target) -> Result
lock(Client, Target, Timeout) -> Result
```

Types:

```
Client = client()
Target = netconf_db()
Timeout = timeout()
Result = ok | {error, error_reason()}
```

Locks the configuration target.

Which target parameters that can be used depends on if `:candidate` and/or `:startup` are supported by the server. If successfull, the configuration system of the device is unavailable to other clients (NETCONF, CORBA, SNMP, and so on). Locks are intended to be short-lived.

Operation *kill_session/2,3* can be used to force the release of a lock owned by another NETCONF session. How this is achieved by the server side is implementation-specific.

```
only_open(Options) -> Result
```

Types:

```
Options = [option()]
Result = {ok, handle()} | {error, error_reason()}
```

Opens a NETCONF session, but does not send `hello`.

As *open/1*, but does not send a `hello` message.

```
only_open(KeyOrName, ExtraOptions) -> Result
```

Types:

```
KeyOrName = ct:key_or_name()
ExtraOptions = [option()]
Result = {ok, handle()} | {error, error_reason()}
```

Opens a named NETCONF session, but does not send `hello`.

As *open/2*, but does not send a `hello` message.

```
open(Options) -> Result
```

Types:

```
Options = [option()]
Result = {ok, handle()} | {error, error_reason()}
```

Opens a NETCONF session and exchanges `hello` messages.

If the server options are specified in a configuration file, or if a named client is needed for logging purposes (see section *Logging* in this module), use *open/2* instead.

The opaque *handle()* reference returned from this function is required as client identifier when calling any other function in this module.

```
open(KeyOrName, ExtraOption) -> Result
```

Types:

```
KeyOrName = ct:key_or_name()
ExtraOption = [option()]
Result = {ok, handle()} | {error, error_reason()}
```

Opens a named NETCONF session and exchanges `hello` messages.

If KeyOrName is a configured server_id() or a target_name() associated with such an Id, then the options for this server are fetched from the configuration file.

The options list is added to those of the configuration file. If an option is specified in both lists, the configuration file take precedence.

If the server is not specified in a configuration file, use *open/1* instead.

The opaque *handle()* reference returned from this function can be used as client identifier when calling any other function in this module. However, if KeyOrName is a target_name(), that is, if the server is named through a call to *ct:require/2* or a require statement in the test suite, then this name can be used instead of *handle()*.

See also *ct:require/2*.

```
send(Client, SimpleXml) -> Result
send(Client, SimpleXml, Timeout) -> Result
```

Types:

```
Client = client()
SimpleXml = simple_xml()
Timeout = timeout()
Result = simple_xml() | {error, error_reason()}
```

Sends an XML document to the server.

The specified XML document is sent "as is" to the server. This function can be used for sending XML documents that cannot be expressed by other interface functions in this module.

```
send_rpc(Client, SimpleXml) -> Result
send_rpc(Client, SimpleXml, Timeout) -> Result
```

Types:

```
Client = client()
SimpleXml = simple_xml()
Timeout = timeout()
Result = [simple_xml()] | {error, error_reason()}
```

Sends a NETCONF rpc request to the server.

The specified XML document is wrapped in a valid NETCONF rpc request and sent to the server. The message-id and namespace attributes are added to element rpc.

This function can be used for sending rpc requests that cannot be expressed by other interface functions in this module.

```
session(Conn) -> Result
session(Conn, Options) -> Result
session(KeyOrName, Conn) -> Result
session(KeyOrName, Conn, Options) -> Result
```

Types:

```
Conn = handle()
Options = [session_option()]
KeyOrName = ct:key_or_name()
Result = {ok, handle()} | {error, error_reason()}
session_option() =
    {timeout, timeout()} | {capability, string() | [string()]}
```

Opens a NETCONF session as a channel on the given SSH connection, and exchanges hello messages with the server.

The opaque *handle()* reference returned from this function can be used as client identifier when calling any other function in this module. However, if KeyOrName is used and it is a target_name(), that is, if the server is named through a call to *ct:require/2* or a require statement in the test suite, then this name can be used instead of *handle()*.

```
unlock(Client, Target) -> Result
unlock(Client, Target, Timeout) -> Result
```

Types:

```
Client = client()
Target = netconf_db()
Timeout = timeout()
Result = ok | {error, error_reason()}
```

Unlocks the configuration target.

If the client earlier has acquired a lock through *lock/2,3*, this operation releases the associated lock. To access another target than running, the server must support :candidate and/or :startup.

ct_rpc

Erlang module

`Common Test` specific layer on Erlang/OTP `rpc`.

Exports

`app_node(App, Candidates) -> NodeName`

Types:

`App = atom()`
`Candidates = [NodeName]`
`NodeName = atom()`

From a set of candidate nodes determines which of them is running the application `App`. If none of the candidate nodes is running `App`, the function makes the test case calling this function to fail. This function is the same as calling `app_node(App, Candidates, true)`.

`app_node(App, Candidates, FailOnBadRPC) -> NodeName`

Types:

`App = atom()`
`Candidates = [NodeName]`
`NodeName = atom()`
`FailOnBadRPC = true | false`

Same as *`ct_rpc:app_node/2`*, except that argument `FailOnBadRPC` determines if the search for a candidate node is to stop if `badrpc` is received at some point.

`app_node(App, Candidates, FailOnBadRPC, Cookie) -> NodeName`

Types:

`App = atom()`
`Candidates = [NodeName]`
`NodeName = atom()`
`FailOnBadRPC = true | false`
`Cookie = atom()`

Same as *`ct_rpc:app_node/2`*, except that argument `FailOnBadRPC` determines if the search for a candidate node is to stop if `badrpc` is received at some point.

The cookie on the client node is set to `Cookie` for this `rpc` operation (used to match the server node cookie).

`call(Node, Module, Function, Args) -> term() | {badrpc, Reason}`

Same as `call(Node, Module, Function, Args, infinity)`.

`call(Node, Module, Function, Args, TimeOut) -> term() | {badrpc, Reason}`

Types:

`Node = NodeName | {Fun, FunArgs}`
`Fun = function()`

FunArgs = term()
NodeName = atom()
Module = atom()
Function = atom()
Args = [term()]
Reason = timeout | term()

Evaluates `apply(Module, Function, Args)` on the node `Node`. Returns either whatever `Function` returns, or `{badrpc, Reason}` if the remote procedure call fails. If `Node` is `{Fun, FunArgs}`, applying `Fun` to `FunArgs` is to return a node name.

`call(Node, Module, Function, Args, TimeOut, Cookie) -> term() | {badrpc, Reason}`

Types:

Node = NodeName | {Fun, FunArgs}
Fun = function()
FunArgs = term()
NodeName = atom()
Module = atom()
Function = atom()
Args = [term()]
Reason = timeout | term()
Cookie = atom()

Evaluates `apply(Module, Function, Args)` on the node `Node`. Returns either whatever `Function` returns, or `{badrpc, Reason}` if the remote procedure call fails. If `Node` is `{Fun, FunArgs}`, applying `Fun` to `FunArgs` is to return a node name.

The cookie on the client node is set to `Cookie` for this `rpc` operation (used to match the server node cookie).

`cast(Node, Module, Function, Args) -> ok`

Types:

Node = NodeName | {Fun, FunArgs}
Fun = function()
FunArgs = term()
NodeName = atom()
Module = atom()
Function = atom()
Args = [term()]
Reason = timeout | term()

Evaluates `apply(Module, Function, Args)` on the node `Node`. No response is delivered and the process that makes the call is not suspended until the evaluation is completed as in the case of `call/3,4`. If `Node` is `{Fun, FunArgs}`, applying `Fun` to `FunArgs` is to return a node name.

`cast(Node, Module, Function, Args, Cookie) -> ok`

Types:

Node = NodeName | {Fun, FunArgs}

```
Fun = function()
FunArgs = term()
NodeName = atom()
Module = atom()
Function = atom()
Args = [term()]
Reason = timeout | term()
Cookie = atom()
```

Evaluates `apply(Module, Function, Args)` on the node `Node`. No response is delivered and the process that makes the call is not suspended until the evaluation is completed as in the case of `call/3,4`. If `Node` is `{Fun, FunArgs}`, applying `Fun` to `FunArgs` is to return a node name.

The cookie on the client node is set to `Cookie` for this `rpc` operation (used to match the server node cookie).

ct_snmp

Erlang module

`Common Test` user interface module for the `SNMP` application.

The purpose of this module is to simplify SNMP configuration for the test case writer. Many test cases can use default values for common operations and then no SNMP configuration files need to be supplied. When it is necessary to change particular configuration parameters, a subset of the relevant SNMP configuration files can be passed to `ct_snmp` by `Common Test` configuration files. For more specialized configuration parameters, a simple SNMP configuration file can be placed in the test suite data directory. To simplify the test suite, `Common Test` keeps track of some of the SNMP manager information. This way the test suite does not have to handle as many input parameters as if it had to interface wthe OTP SNMP manager directly.

Configurable SNMP Manager and Agent Parameters:

Manager configuration:

`[{start_manager, boolean()}`

Optional. Default is `true`.

`{users, [{user_name(), [call_back_module(), user_data()]}]}`

Optional.

`{usm_users, [{usm_user_name(), [usm_config()]}]}`

Optional. SNMPv3 only.

`{managed_agents,[{agent_name(), [user_name(), agent_ip(), agent_port(), [agent_config()]]}]}`

`managed_agents` is optional.

`{max_msg_size, integer()}`

Optional. Default is `484`.

`{mgr_port, integer()}`

Optional. Default is `5000`.

`{engine _id, string()}`

Optional. Default is `"mgrEngine"`.

Agent configuration:

`{start_agent, boolean()}`

Optional. Default is `false`.

`{agent_sysname, string()}`

Optional. Default is `"ct_test"`.

`{agent_manager_ip, manager_ip()}`

Optional. Default is `localhost`.

`{agent_vsns, list()}`

Optional. Default is `[v2]`.

`{agent_trap_udp, integer()}`

Optional. Default is `5000`.

`{agent_udp, integer()}`

Optional. Default is `4000`.

`{agent_notify_type, atom()}`

Optional. Default is `trap`.

`{agent_sec_type, sec_type()}`

Optional. Default is `none`.

`{agent_passwd, string()}`

Optional. Default is `""`.

`{agent_engine_id, string()}`

Optional. Default is `"agentEngine"`.

`{agent_max_msg_size, string()}`

Optional. Default is `484`.

The following parameters represents the SNMP configuration files `context.conf`, `standard.conf`, `community.conf`, `vacm.conf`, `usm.conf`, `notify.conf`, `target_addr.conf`, and `target_params.conf`. Notice that all values in `agent.conf` can be modified by the parameters listed above. All these configuration files have default values set by the `SNMP` application. These values can be overridden by suppling a list of valid configuration values or a file located in the test suites data directory, which can produce a list of valid configuration values if you apply function `file:consult/1` to the file.

`{agent_contexts, [term()] | {data_dir_file, rel_path()}}`

Optional.

`{agent_community, [term()] | {data_dir_file, rel_path()}}`

Optional.

`{agent_sysinfo, [term()] | {data_dir_file, rel_path()}}`

Optional.

`{agent_vacm, [term()] | {data_dir_file, rel_path()}}`

Optional.

`{agent_usm, [term()] | {data_dir_file, rel_path()}}`

Optional.

`{agent_notify_def, [term()] | {data_dir_file, rel_path()}}`

Optional.

`{agent_target_address_def, [term()] | {data_dir_file, rel_path()}}`

Optional.

`{agent_target_param_def, [term()] | {data_dir_file, rel_path()}}`

Optional.

Parameter `MgrAgentConfName` in the functions is to be a name you allocate in your test suite using a `require` statement. Example (where `MgrAgentConfName = snmp_mgr_agent`):

```
suite() -> [{require, snmp_mgr_agent, snmp}].
```

or

```
ct:require(snmp_mgr_agent, snmp).
```

Notice that USM users are needed for SNMPv3 configuration and are not to be confused with users.

SNMP traps, inform, and report messages are handled by the user callback module. For details, see the *SNMP* application.

It is recommended to use the `.hrl` files created by the Erlang/OTP MIB compiler to define the Object Identifiers (OIDs). For example, to get the Erlang node name from `erlNodeTable` in the OTP-MIB:

```
Oid = ?erlNodeEntry ++ [?erlNodeName, 1]
```

Furthermore, values can be set for SNMP application configuration parameters, `config`, `server`, `net_if`, and so on (for a list of valid parameters and types, see the *User's Guide for the SNMP application*). This is done by defining a configuration data variable on the following form:

```
{snmp_app, [{manager, [snmp_app_manager_params()]},
            {agent, [snmp_app_agent_params()]}]}.
```

A name for the data must be allocated in the suite using `require` (see the example above). Pass this name as argument `SnmpAppConfName` to *`ct_snmp:start/3`*. `ct_snmp` specifies default values for some SNMP application configuration parameters (such as `{verbosity,trace}` for parameter `config`). This set of defaults is merged with the parameters specified by the user. The user values override `ct_snmp` defaults.

Data Types

```
agent_config() = {Item, Value}
agent_ip() = ip()
agent_name() = atom()
agent_port() = integer()
call_back_module() = atom()
error_index() = integer()
error_status() = noError | atom()
ip() = string() | {integer(), integer(), integer(), integer()}
manager_ip() = ip()
oid() = [byte()]
oids() = [oid()]
rel_path() = string()
sec_type() = none | minimum | semi
snmp_app_agent_params() = term()
snmp_app_manager_params() = term()
snmpreply() = {error_status(), error_index(), varbinds()}
user_data() = term()
user_name() = atom()
usm_config() = {Item, Value}
usm_user_name() = string()
value_type() = o('OBJECT IDENTIFIER') | i('INTEGER') | u('Unsigned32') |
g('Unsigned32') | s('OCTET STRING')
var_and_val() = {oid(), value_type(), value()}
```

```
varbind() = term()
varbinds() = [varbind()]
varsandvals() = [var_and_val()]
```

These data types are described in the documentation for the *SNMP* application.

Exports

`get_next_values(Agent, Oids, MgrAgentConfName) -> SnmpReply`

Types:

`Agent = agent_name()`
`Oids = oids()`
`MgrAgentConfName = atom()`
`SnmpReply = snmpreply()`

Issues a synchronous SNMP `get next` request.

`get_values(Agent, Oids, MgrAgentConfName) -> SnmpReply`

Types:

`Agent = agent_name()`
`Oids = oids()`
`MgrAgentConfName = atom()`
`SnmpReply = snmpreply()`

Issues a synchronous SNMP `get` request.

`load_mibs(Mibs) -> ok | {error, Reason}`

Types:

`Mibs = [MibName]`
`MibName = string()`
`Reason = term()`

Loads the MIBs into agent `snmp_master_agent`.

`register_agents(MgrAgentConfName, ManagedAgents) -> ok | {error, Reason}`

Types:

`MgrAgentConfName = atom()`
`ManagedAgents = [agent()]`
`Reason = term()`

Explicitly instructs the manager to handle this agent. Corresponds to making an entry in `agents.conf`.

This function tries to register the specified managed agents, without checking if any of them exist. To change a registered managed agent, the agent must first be unregistered.

`register_users(MgrAgentConfName, Users) -> ok | {error, Reason}`

Types:

`MgrAgentConfName = atom()`
`Users = [user()]`

Reason = term()

Registers the manager entity (=user) responsible for specific agent(s). Corresponds to making an entry in `users.conf`.

This function tries to register the specified users, without checking if any of them exist. To change a registered user, the user must first be unregistered.

```
register_usm_users(MgrAgentConfName, UsmUsers) -> ok | {error, Reason}
```

Types:

MgrAgentConfName = atom()
UsmUsers = [usm_user()]
Reason = term()

Explicitly instructs the manager to handle this USM user. Corresponds to making an entry in `usm.conf`.

This function tries to register the specified users, without checking if any of them exist. To change a registered user, the user must first be unregistered.

```
set_info(Config) -> [{Agent, OldVarsAndVals, NewVarsAndVals}]
```

Types:

Config = [{Key, Value}]
Agent = agent_name()
OldVarsAndVals = varsandvals()
NewVarsAndVals = varsandvals()

Returns a list of all successful `set` requests performed in the test case in reverse order. The list contains the involved user and agent, the value before `set`, and the new value. This is intended to simplify the cleanup in function `end_per_testcase`, that is, the undoing of the `set` requests and their possible side-effects.

```
set_values(Agent, VarsAndVals, MgrAgentConfName, Config) -> SnmpReply
```

Types:

Agent = agent_name()
Oids = oids()
MgrAgentConfName = atom()
Config = [{Key, Value}]
SnmpReply = snmpreply()

Issues a synchronous SNMP `set` request.

```
start(Config, MgrAgentConfName) -> ok
```

Equivalent to *`ct_snmp:start(Config, MgrAgentConfName, undefined)`*.

```
start(Config, MgrAgentConfName, SnmpAppConfName) -> ok
```

Types:

Config = [{Key, Value}]
Key = atom()
Value = term()
MgrAgentConfName = atom()

SnmpConfName = atom()

Starts an SNMP manager and/or agent. In the manager case, registrations of users and agents, as specified by the configuration `MgrAgentConfName`, are performed. When using SNMPv3, called USM users are also registered. Users, `usm_users`, and managed agents can also be registered later using *`ct_snmp:register_users/2`*, *`ct_snmp:register_agents/2`*, and *`ct_snmp:register_usm_users/2`*.

The agent started is called `snmp_master_agent`. Use *`ct_snmp:load_mibs/1`* to load MIBs into the agent.

With `SnmpAppConfName` SNMP applications can be configured with parameters `config`, `mibs`, `net_if`, and so on. The values are merged with (and possibly override) default values set by `ct_snmp`.

`stop(Config) -> ok`

Types:

Config = [{Key, Value}]
Key = atom()
Value = term()

Stops the SNMP manager and/or agent, and removes all files created.

`unload_mibs(Mibs) -> ok | {error, Reason}`

Types:

Mibs = [MibName]
MibName = string()
Reason = term()

Unloads the MIBs from agent `snmp_master_agent`.

`unregister_agents(MgrAgentConfName) -> ok`

Types:

MgrAgentConfName = atom()
Reason = term()

Unregisters all managed agents.

`unregister_agents(MgrAgentConfName, ManagedAgents) -> ok`

Types:

MgrAgentConfName = atom()
ManagedAgents = [agent_name()]
Reason = term()

Unregisters the specified managed agents.

`unregister_users(MgrAgentConfName) -> ok`

Types:

MgrAgentConfName = atom()
Reason = term()

Unregisters all users.

```
unregister_users(MgrAgentConfName, Users) -> ok
```

Types:

MgrAgentConfName = atom()
Users = [user_name()]
Reason = term()

Unregisters the specified users.

```
unregister_usm_users(MgrAgentConfName) -> ok
```

Types:

MgrAgentConfName = atom()
Reason = term()

Unregisters all USM users.

```
unregister_usm_users(MgrAgentConfName, UsmUsers) -> ok
```

Types:

MgrAgentConfName = atom()
UsmUsers = [usm_user_name()]
Reason = term()

Unregisters the specified USM users.

ct_telnet

Erlang module

`Common Test` specific layer on top of Telnet client `ct_telnet_client.erl`.

Use this module to set up Telnet connections, send commands, and perform string matching on the result. For information about how to use `ct_telnet` and configure connections, specifically for UNIX hosts, see the *`unix_telnet`* manual page.

Default values defined in `ct_telnet`:

- Connection timeout (time to wait for connection) = 10 seconds
- Command timeout (time to wait for a command to return) = 10 seconds
- Max number of reconnection attempts = 3
- Reconnection interval (time to wait in between reconnection attempts) = 5 seconds
- Keep alive (sends NOP to the server every 8 sec if connection is idle) = `true`
- Polling limit (max number of times to poll to get a remaining string terminated) = 0
- Polling interval (sleep time between polls) = 1 second
- The TCP_NODELAY option for the telnet socket is disabled (set to `false`) per default

These parameters can be modified by the user with the following configuration term:

```
{telnet_settings, [{connect_timeout,Millisec},
                   {command_timeout,Millisec},
                   {reconnection_attempts,N},
                   {reconnection_interval,Millisec},
                   {keep_alive,Bool},
                   {poll_limit,N},
                   {poll_interval,Millisec},
                   {tcp_nodelay,Bool}]}.
```

`Millisec = integer(), N = integer()`

Enter the `telnet_settings` term in a configuration file included in the test and `ct_telnet` retrieves the information automatically.

`keep_alive` can be specified per connection, if necessary. For details, see *`unix_telnet`*.

Logging

The default logging behavior of `ct_telnet` is to print information about performed operations, commands, and their corresponding results to the test case HTML log. The following is not printed to the HTML log: text strings sent from the Telnet server that are not explicitly received by a `ct_telnet` function, such as `expect/3`. However, `ct_telnet` can be configured to use a special purpose event handler, implemented in `ct_conn_log_h`, for logging **all** Telnet traffic. To use this handler, install a `Common Test` hook named `cth_conn_log`. Example (using the test suite information function):

```
suite() ->
    [{ct_hooks, [{cth_conn_log, [{conn_mod(),hook_options()}]}]}].
```

`conn_mod()` is the name of the `Common Test` module implementing the connection protocol, that is, `ct_telnet`.

The `cth_conn_log` hook performs unformatted logging of Telnet data to a separate text file. All Telnet communication is captured and printed, including any data sent from the server. The link to this text file is located at the top of the test case HTML log.

By default, data for all Telnet connections is logged in one common file (named `default`), which can get messy, for example, if multiple Telnet sessions are running in parallel. Therefore a separate log file can be created for each connection. To configure this, use hook option `hosts` and list the names of the servers/connections to be used in the suite. The connections must be named for this to work (see *`ct_telnet:open/1,2,3,4`*).

Hook option `log_type` can be used to change the `cth_conn_log` behavior. The default value of this option is `raw`, which results in the behavior described above. If the value is set to `html`, all Telnet communication is printed to the test case HTML log instead.

All `cth_conn_log` hook options described can also be specified in a configuration file with configuration variable `ct_conn_log`.

Example:

```
{ct_conn_log, [{ct_telnet,[{log_type,raw},
                           {hosts,[key_or_name()]}]}]}
```

Note:

Hook options specified in a configuration file overwrite any hard-coded hook options in the test suite.

Logging Example:

The following `ct_hooks` statement causes printing of Telnet traffic to separate logs for the connections `server1` and `server2`. Traffic for any other connections is logged in the default Telnet log.

```
suite() ->
    [{ct_hooks,
      [{cth_conn_log, [{ct_telnet,[{hosts,[server1,server2]}]}]}]}].
```

As previously explained, this specification can also be provided by an entry like the following in a configuration file:

```
{ct_conn_log, [{ct_telnet,[{hosts,[server1,server2]}]}]}.
```

In this case the `ct_hooks` statement in the test suite can look as follows:

```
suite() ->
    [{ct_hooks, [{cth_conn_log, []}]}].
```

Data Types

`connection() = handle() | {target_name(), connection_type()} | target_name()`

For `target_name()`, see module *`ct`*.

`connection_type() = telnet | ts1 | ts2`
`handle() = handle()`

Handle for a specific Telnet connection, see module *`ct`*.

`prompt_regexp() = string()`

Regular expression matching all possible prompts for a specific target type. `regexp` must not have any groups, that is, when matching, `re:run/3` (in STDLIB) must return a list with one single element.

Exports

`close(Connection) -> ok | {error, Reason}`

Types:

`Connection = connection()`
`Reason = term()`

Closes the Telnet connection and stops the process managing it.

A connection can be associated with a target name and/or a handle. If `Connection` has no associated target name, it can only be closed with the handle value (see *`ct_telnet:open/4`*).

`cmd(Connection, Cmd) -> {ok, Data} | {error, Reason}`

Equivalent to *`ct_telnet:cmd(Connection, Cmd, [])`*.

`cmd(Connection, Cmd, Opts) -> {ok, Data} | {error, Reason}`

Types:

`Connection = connection()`
`Cmd = string()`
`Opts = [Opt]`
`Opt = {timeout, timeout()} | {newline, boolean() | string()}`
`Data = [string()]`
`Reason = term()`

Sends a command through Telnet and waits for prompt.

By default, this function adds "\n" to the end of the specified command. If this is not desired, use option `{newline,false}`. This is necessary, for example, when sending Telnet command sequences prefixed with character Interpret As Command (IAC). Option `{newline,string()}` can also be used if a different line end than "\n" is required, for instance `{newline,"\r\n"}`, to add both carriage return and newline characters.

Option `timeout` specifies how long the client must wait for prompt. If the time expires, the function returns `{error,timeout}`. For information about the default value for the command timeout, see the *list of default values* in the beginning of this module.

`cmdf(Connection, CmdFormat, Args) -> {ok, Data} | {error, Reason}`

Equivalent to *`ct_telnet:cmdf(Connection, CmdFormat, Args, [])`*.

`cmdf(Connection, CmdFormat, Args, Opts) -> {ok, Data} | {error, Reason}`

Types:

`Connection = connection()`
`CmdFormat = string()`
`Args = list()`
`Opts = [Opt]`
`Opt = {timeout, timeout()} | {newline, boolean() | string()}`

Data = [string()]
Reason = term()

Sends a Telnet command and waits for prompt (uses a format string and a list of arguments to build the command).

For details, see *ct_telnet:cmd/3*.

`expect(Connection, Patterns) -> term()`

Equivalent to *ct_telnet:expect(Connections, Patterns, [])*.

`expect(Connection, Patterns, Opts) -> {ok, Match} | {ok, MatchList, HaltReason} | {error, Reason}`

Types:

Connection = connection()
Patterns = Pattern | [Pattern]
Pattern = string() | {Tag, string()} | prompt | {prompt, Prompt}
Prompt = string()
Tag = term()
Opts = [Opt]
Opt = {idle_timeout, IdleTimeout} | {total_timeout, TotalTimeout} | repeat | {repeat, N} | sequence | {halt, HaltPatterns} | ignore_prompt | no_prompt_check | wait_for_prompt | {wait_for_prompt, Prompt}
IdleTimeout = infinity | integer()
TotalTimeout = infinity | integer()
N = integer()
HaltPatterns = Patterns
MatchList = [Match]
Match = RxMatch | {Tag, RxMatch} | {prompt, Prompt}
RxMatch = [string()]
HaltReason = done | Match
Reason = timeout | {prompt, Prompt}

Gets data from Telnet and waits for the expected pattern.

`Pattern` can be a POSIX regular expression. The function returns when a pattern is successfully matched (at least one, in the case of multiple patterns).

`RxMatch` is a list of matched strings. It looks as follows `[FullMatch, SubMatch1, SubMatch2, ...]`, where `FullMatch` is the string matched by the whole regular expression, and `SubMatchN` is the string that matched subexpression number `N`. Subexpressions are denoted with `'(' ')'` in the regular expression.

If a `Tag` is specified, the returned `Match` also includes the matched `Tag`. Otherwise, only `RxMatch` is returned.

Options:

`idle_timeout`

Indicates that the function must return if the Telnet client is idle (that is, if no data is received) for more than `IdleTimeout` milliseconds. Default time-out is 10 seconds.

`total_timeout`

Sets a time limit for the complete `expect` operation. After `TotalTimeout` milliseconds, `{error,timeout}` is returned. Default is `infinity` (that is, no time limit).

`ignore_prompt | no_prompt_check`

>The function returns when a prompt is received, even if no pattern has yet been matched, and `{error, {prompt,Prompt}}` is returned. However, this behavior can be modified with option `ignore_prompt` or option `no_prompt_check`, which tells `expect` to return only when a match is found or after a time-out.

`ignore_prompt`

`ct_telnet` ignores any prompt found. This option is useful if data sent by the server can include a pattern matching prompt `regexp` (as returned by `TargedMod:get_prompt_regexp/0`), but is not to not cause the function to return.

`no_prompt_check`

`ct_telnet` does not search for a prompt at all. This is useful if, for example, `Pattern` itself matches the prompt.

`wait_for_prompt`

Forces `ct_telnet` to wait until the prompt string is received before returning (even if a pattern has already been matched). This is equal to calling `expect(Conn, Patterns++[{prompt,Prompt}], [sequence|Opts])`. Notice that option `idle_timeout` and `total_timeout` can abort the operation of waiting for prompt.

`repeat | repeat, N`

The pattern(s) must be matched multiple times. If `N` is specified, the pattern(s) are matched `N` times, and the function returns `HaltReason = done`. This option can be interrupted by one or more `HaltPatterns`. `MatchList` is always returned, that is, a list of `Match` instead of only one `Match`. Also `HaltReason` is returned.

`sequence`

All patterns must be matched in a sequence. A match is not concluded until all patterns are matched. This option can be interrupted by one or more `HaltPatterns`. `MatchList` is always returned, that is, a list of `Match` instead of only one `Match`. Also `HaltReason` is returned.

Example 1:

```
expect(Connection,[{abc,"ABC"},{xyz,"XYZ"}],[sequence,{halt,[{nnn,"NNN"}]}])
```

First this tries to match `"ABC"`, and then `"XYZ"`, but if `"NNN"` appears, the function returns `{error,{nnn,["NNN"]}}`. If both `"ABC"` and `"XYZ"` are matched, the function returns `{ok,[AbcMatch,XyzMatch]}`.

Example 2:

```
expect(Connection,[{abc,"ABC"},{xyz,"XYZ"}],[{repeat,2},{halt,[{nnn,"NNN"}]}])
```

This tries to match `"ABC"` or `"XYZ"` twice. If `"NNN"` appears, the function returns `HaltReason = {nnn,["NNN"]}`.

Options `repeat` and `sequence` can be combined to match a sequence multiple times.

`get_data(Connection) -> {ok, Data} | {error, Reason}`

Types:

`Connection = connection()`

`Data = [string()]`

`Reason = term()`

Gets all data received by the Telnet client since the last command was sent. Only newline-terminated strings are returned. If the last received string has not yet been terminated, the connection can be polled automatically until the string is complete.

The polling feature is controlled by the configuration values `poll_limit` and `poll_interval` and is by default disabled. This means that the function immediately returns all complete strings received and saves a remaining non-terminated string for a later `get_data` call.

`open(Name) -> {ok, Handle} | {error, Reason}`

Equivalent to *`ct_telnet:open(Name, telnet)`*.

`open(Name, ConnType) -> {ok, Handle} | {error, Reason}`

Types:

`Name = target_name()`
`ConnType = connection_type()`
`Handle = handle()`
`Reason = term()`

Opens a Telnet connection to the specified target host.

`open(KeyOrName, ConnType, TargetMod) -> {ok, Handle} | {error, Reason}`

Equivalent to *`ct_telnet:ct_telnet:open(KeyOrName, ConnType, TargetMod, [])`*.

`open(KeyOrName, ConnType, TargetMod, Extra) -> {ok, Handle} | {error, Reason}`

Types:

`KeyOrName = Key | Name`
`Key = atom()`
`Name = target_name()`
`ConnType = connection_type()`
`TargetMod = atom()`
`Extra = term()`
`Handle = handle()`
`Reason = term()`

Opens a Telnet connection to the specified target host.

The target data must exist in a configuration file. The connection can be associated with `Name` and/or the returned `Handle`. To allocate a name for the target, use one of the following alternatives:

- *`ct:require/2`* in a test case
- A `require` statement in the suite information function (`suite/0`)
- A `require` statement in a test case information function

If you want the connection to be associated with `Handle` only (if you, for example, need to open multiple connections to a host), use `Key`, the configuration variable name, to specify the target. Notice that a connection without an associated target name can only be closed with the `Handle` value.

`TargetMod` is a module that exports the functions `connect(Ip, Port, KeepAlive, Extra)` and `get_prompt_regexp()` for the specified `TargetType` (for example, `unix_telnet`).

For `target_name()`, see module *`ct`*.

See also *ct:require/2*.

`send(Connection, Cmd) -> ok | {error, Reason}`

Equivalent to *ct_telnet:send(Connection, Cmd, [])*.

`send(Connection, Cmd, Opts) -> ok | {error, Reason}`

Types:

Connection = connection()
Cmd = string()
Opts = [Opt]
Opt = {newline, boolean() | string()}
Reason = term()

Sends a Telnet command and returns immediately.

By default, this function adds "\n" to the end of the specified command. If this is not desired, option `{newline,false}` can be used. This is necessary, for example, when sending Telnet command sequences prefixed with character Interpret As Command (IAC). Option `{newline,string()}` can also be used if a different line end than "\n" is required, for instance `{newline,"\r\n"}`, to add both carriage return and newline characters.

The resulting output from the command can be read with *ct_telnet:get_data/2* or *ct_telnet:expect/2,3*.

`sendf(Connection, CmdFormat, Args) -> ok | {error, Reason}`

Equivalent to *ct_telnet:sendf(Connection, CmdFormat, Args, [])*.

`sendf(Connection, CmdFormat, Args, Opts) -> ok | {error, Reason}`

Types:

Connection = connection()
CmdFormat = string()
Args = list()
Opts = [Opt]
Opt = {newline, boolean() | string()}
Reason = term()

Sends a Telnet command and returns immediately (uses a format string and a list of arguments to build the command).

For details, see *ct_telnet:send/3*.

See Also

unix_telnet

unix_telnet

Erlang module

Callback module for *ct_telnet*, for connecting to a Telnet server on a UNIX host.

It requires the following entry in the configuration file:

```
{unix,[{telnet,HostNameOrIpAddress},
       {port,PortNum},                % optional
       {username,UserName},
       {password,Password},
       {keep_alive,Bool}]}.           % optional
```

To communicate through Telnet to the host specified by `HostNameOrIpAddress`, use the interface functions in *ct_telnet*, for example, `open(Name)` and `cmd(Name,Cmd)`.

`Name` is the name you allocated to the Unix host in your `require` statement, for example:

```
suite() -> [{require,Name,{unix,[telnet]}}].
```

or

```
ct:require(Name,{unix,[telnet]}).
```

The "keep alive" activity (that is, that `Common Test` sends NOP to the server every 10 seconds if the connection is idle) can be enabled or disabled for one particular connection as described here. It can be disabled for all connections using `telnet_settings` (see *ct_telnet*).

The `{port,PortNum}` tuple is optional and if omitted, default Telnet port 23 is used. Also the `keep_alive` tuple is optional, and the value defauls to `true` (enabled).

Exports

`connect(ConnName, Ip, Port, Timeout, KeepAlive, TCPNoDelay, Extra) -> {ok, Handle} | {error, Reason}`

Types:

`ConnName = target_name()`
`Ip = string() | {integer(), integer(), integer(), integer()}`
`Port = integer()`
`Timeout = integer()`
`KeepAlive = bool()`
`TCPNoDelay = bool()`
`Extra = target_name() | {Username, Password}`
`Username = string()`
`Password = string()`
`Handle = handle()`
`Reason = term()`

Callback for `ct_telnet.erl`.

Setup Telnet connection to a Unix host.

For `target_name()`, see *`ct`*. For `handle()`, see *`ct_telnet`*.

`get_prompt_regexp() -> PromptRegexp`

Types:

`PromptRegexp = prompt_regexp()`

Callback for `ct_telnet.erl`.

Returns a suitable `regexp` string matching common prompts for users on Unix hosts.

For `prompt_regexp()`, see *`ct_telnet`*.

See Also

`ct`, *`ct_telnet`*

ct_slave

Erlang module

`Common Test` framework functions for starting and stopping nodes for Large-Scale Testing.

This module exports functions used by the `Common Test` Master to start and stop "slave" nodes. It is the default callback module for the `{init, node_start}` term in the Test Specification.

Exports

`start(Node) -> Result`

Types:

```
Node = atom()
Result = {ok, NodeName} | {error, Reason, NodeName}
Reason = already_started | started_not_connected | boot_timeout |
init_timeout | startup_timeout | not_alive
NodeName = atom()
```

Starts an Erlang node with name `Node` on the local host.

See also *`ct_slave:start/3`*.

`start(HostOrNode, NodeOrOpts) -> Result`

Types:

```
HostOrNode = atom()
NodeOrOpts = atom() | list()
Result = {ok, NodeName} | {error, Reason, NodeName}
Reason = already_started | started_not_connected | boot_timeout |
init_timeout | startup_timeout | not_alive
NodeName = atom()
```

Starts an Erlang node with default options on a specified host, or on the local host with specified options. That is, the call is interpreted as `start(Host, Node)` when the second argument is atom-valued and `start(Node, Opts)` when it is list-valued.

See also *`ct_slave:start/3`*.

`start(Host, Node, Opts) -> Result`

Types:

```
Node = atom()
Host = atom()
Opts = [OptTuples]
OptTuples = {username, Username} | {password, Password} | {boot_timeout,
BootTimeout} | {init_timeout, InitTimeout} | {startup_timeout,
StartupTimeout} | {startup_functions, StartupFunctions} | {monitor_master,
Monitor} | {kill_if_fail, KillIfFail} | {erl_flags, ErlangFlags} | {env,
[{EnvVar, Value}]}
Username = string()
```

```
Password = string()
BootTimeout = integer()
InitTimeout = integer()
StartupTimeout = integer()
StartupFunctions = [StartupFunctionSpec]
StartupFunctionSpec = {Module, Function, Arguments}
Module = atom()
Function = atom()
Arguments = [term]
Monitor = bool()
KillIfFail = bool()
ErlangFlags = string()
EnvVar = string()
Value = string()
Result = {ok, NodeName} | {error, Reason, NodeName}
Reason = already_started | started_not_connected | boot_timeout |
init_timeout | startup_timeout | not_alive
NodeName = atom()
```

Starts an Erlang node with name `Node` on host `Host` as specified by the combination of options in `Opts`.

Options `Username` and `Password` are used to log on to the remote host `Host`. `Username`, if omitted, defaults to the current username. `Password` is empty by default.

A list of functions specified in option `Startup` are executed after startup of the node. Notice that all used modules are to be present in the code path on `Host`.

The time-outs are applied as follows:

`BootTimeout`

The time to start the Erlang node, in seconds. Defaults to 3 seconds. If the node is not pingable within this time, the result `{error, boot_timeout, NodeName}` is returned.

`InitTimeout`

The time to wait for the node until it calls the internal callback function informing master about a successful startup. Defaults to 1 second. In case of a timed out message, the result `{error, init_timeout, NodeName}` is returned.

`StartupTimeout`

The time to wait until the node stops to run `StartupFunctions`. Defaults to 1 second. If this time-out occurs, the result `{error, startup_timeout, NodeName}` is returned.

Options:

`monitor_master`

Specifies if the slave node is to be stopped if the master node stops. Defaults to `false`.

`kill_if_fail`

Specifies if the slave node is to be killed if a time-out occurs during initialization or startup. Defaults to `true`. Notice that the node can also be still alive it the boot time-out occurred, but it is not killed in this case.

`erlang_flags`

Specifies which flags are added to the parameters of the executable `erl`.

`env`

Specifies a list of environment variables that will extend the environment.

Special return values:

- `{error, already_started, NodeName}` if the node with the specified name is already started on a specified host.
- `{error, started_not_connected, NodeName}` if the node is started, but not connected to the master node.
- `{error, not_alive, NodeName}` if the node on which *`ct_slave:start/3`* is called, is not alive. Notice that `NodeName` is the name of the current node in this case.

`stop(Node) -> Result`

Types:

```
Node = atom()
Result = {ok, NodeName} | {error, Reason, NodeName}
Reason = not_started | not_connected | stop_timeout
```

Stops the running Erlang node with name `Node` on the local host.

`stop(Host, Node) -> Result`

Types:

```
Host = atom()
Node = atom()
Result = {ok, NodeName} | {error, Reason, NodeName}
Reason = not_started | not_connected | stop_timeout
NodeName = atom()
```

Stops the running Erlang node with name `Node` on host `Host`.

ct_hooks

Erlang module

The **Common Test Hook (CTH)** framework allows extensions of the default behavior of `Common Test` by callbacks before and after all test suite calls. It is intended for advanced users of `Common Test` who want to abstract out behavior that is common to multiple test suites.

In brief, CTH allows you to:

- Manipulate the runtime configuration before each suite configuration call.
- Manipulate the return of all suite configuration calls and by extension the result of the test themselves.

The following sections describe the mandatory and optional CTH functions that `Common Test` calls during test execution. For more details, see section *Common Test Hooks* in the User's Guide.

For information about how to add a CTH to your suite, see section *Installing a CTH* in the User's Guide.

Note:

For a minimal example of a CTH, see section *Example CTH* in the User's Guide.

Callback Functions

The following functions define the callback interface for a CTH.

Exports

`Module:init(Id, Opts) -> {ok, State} | {ok, State, Priority}`

Types:

`Id = reference() | term()`
`Opts = term()`
`State = term()`
`Priority = integer()`

MANDATORY

This function is always called before any other callback function. Use it to initiate any common state. It is to return a state for this CTH.

`Id` is either the return value of *`ct_hooks:id/1`*, or a `reference` (created using *erlang:make_ref/0* in ERTS) if *`ct_hooks:id/1`* is not implemented.

`Priority` is the relative priority of this hook. Hooks with a lower priority are executed first. If no priority is specified, it is set to `0`.

For details about when `init` is called, see section *CTH Scope* in the User's Guide.

`Module:post_groups(SuiteName, GroupDefs) -> NewGroupDefs`

Types:

`SuiteName = atom()`
`GroupDefs = NewGroupDefs = [Group]`
`Group = {GroupName,Properties,GroupsAndTestCases}`

```
GroupName = atom()
Properties = [parallel | sequence | Shuffle | {GroupRepeatType,N}]
GroupsAndTestCases = [Group | {group,GroupName} | TestCase |
{testcase,TestCase,TCRepeatProps}]
TestCase = atom()
TCRepeatProps = [{repeat,N} | {repeat_until_ok,N} | {repeat_until_fail,N}]
Shuffle = shuffle | {shuffle,Seed}
Seed = {integer(),integer(),integer()}
GroupRepeatType = repeat | repeat_until_all_ok | repeat_until_all_fail |
repeat_until_any_ok | repeat_until_any_fail
N = integer() | forever
```

OPTIONAL

This function is called after *`groups/0`*. It is used to modify the test group definitions, for instance to add or remove groups or change group properties.

`GroupDefs` is what *`groups/0`* returned, that is, a list of group definitions.

`NewGroupDefs` is the possibly modified version of this list.

This function is called only if the CTH is added before `init_per_suite` is run. For details, see section *CTH Scope* in the User's Guide.

Notice that for CTHs that are installed by means of the *`suite/0`* function, `post_groups/2` is called before the *`init/2`* hook function. However, for CTHs that are installed by means of the CT start flag, the *`init/2`* function is called first.

Note:

Prior to each test execution, Common Test does a simulated test run in order to count test suites, groups and cases for logging purposes. This causes the `post_groups/2` hook function to always be called twice. For this reason, side effects are best avoided in this callback.

`Module:post_all(SuiteName, Return, GroupDefs) -> NewReturn`

Types:

```
SuiteName = atom()
Return = NewReturn = Tests | {skip,Reason}
Tests = [TestCase | {testcase,TestCase,TCRepeatProps} | {group,GroupName}
| {group,GroupName,Properties} | {group,GroupName,Properties,SubGroups}]
TestCase = atom()
TCRepeatProps = [{repeat,N} | {repeat_until_ok,N} | {repeat_until_fail,N}]
GroupName = atom()
Properties = GroupProperties | default
SubGroups = [{GroupName,Properties} | {GroupName,Properties,SubGroups}]
Shuffle = shuffle | {shuffle,Seed}
Seed = {integer(),integer(),integer()}
GroupRepeatType = repeat | repeat_until_all_ok | repeat_until_all_fail |
repeat_until_any_ok | repeat_until_any_fail
N = integer() | forever
```

```
GroupDefs = NewGroupDefs = [Group]
Group = {GroupName,GroupProperties,GroupsAndTestCases}
GroupProperties = [parallel | sequence | Shuffle | {GroupRepeatType,N}]
GroupsAndTestCases = [Group | {group,GroupName} | TestCase]
Reason = term()
```

OPTIONAL

This function is called after *`all/0`*. It is used to modify the set of test cases and test group to be executed, for instance to add or remove test cases and groups, change group properties, or even skip all tests in the suite.

`Return` is what *`all/0`* returned, that is, a list of test cases and groups to be executed, or a tuple `{skip,Reason}`.

`GroupDefs` is what *`groups/0`* or the `post_groups/2` hook returned, that is, a list of group definitions.

`NewReturn` is the possibly modified version of `Return`.

This function is called only if the CTH is added before `init_per_suite` is run. For details, see section *CTH Scope* in the User's Guide.

Notice that for CTHs that are installed by means of the *`suite/0`* function, `post_all/2` is called before the *`init/2`* hook function. However, for CTHs that are installed by means of the CT start flag, the *`init/2`* function is called first.

Note:

Prior to each test execution, Common Test does a simulated test run in order to count test suites, groups and cases for logging purposes. This causes the `post_all/3` hook function to always be called twice. For this reason, side effects are best avoided in this callback.

```
Module:pre_init_per_suite(SuiteName, InitData, CTHState) -> Result
```

Types:

```
SuiteName = atom()
InitData = Config | SkipOrFail
Config = NewConfig = [{Key,Value}]
CTHState = NewCTHState = term()
Result = {Return, NewCTHState}
Return = NewConfig | SkipOrFail
SkipOrFail = {fail, Reason} | {skip, Reason}
Key = atom()
Value = term()
Reason = term()
```

OPTIONAL

This function is called before *`init_per_suite`* if it exists. It typically contains initialization/logging that must be done before `init_per_suite` is called. If `{skip,Reason}` or `{fail,Reason}` is returned, `init_per_suite` and all test cases of the suite are skipped and `Reason` printed in the overview log of the suite.

`SuiteName` is the name of the suite to be run.

`InitData` is the original configuration list of the test suite, or a `SkipOrFail` tuple if a previous CTH has returned this.

`CTHState` is the current internal state of the CTH.

`Return` is the result of the `init_per_suite` function. If it is `{skip,Reason}` or `{fail,Reason}`, *`init_per_suite`* is never called, instead the initiation is considered to be skipped or failed, respectively. If a `NewConfig` list is returned, *`init_per_suite`* is called with that `NewConfig` list. For more details, see section *Pre Hooks* in the User's Guide.

This function is called only if the CTH is added before `init_per_suite is run`. For details, see section *CTH Scope* in the User's Guide.

```
Module:post_init_per_suite(SuiteName, Config, Return, CTHState) -> Result
```

Types:

```
SuiteName = atom()
Config = [{Key,Value}]
Return = NewReturn = Config | SkipOrFail | term()
SkipOrFail = {fail, Reason} | {skip, Reason} | term()
CTHState = NewCTHState = term()
Result = {NewReturn, NewCTHState}
Key = atom()
Value = term()
Reason = term()
```

OPTIONAL

This function is called after *`init_per_suite`* if it exists. It typically contains extra checks to ensure that all the correct dependencies are started correctly.

`Return` is what *`init_per_suite`* returned, that is, `{fail,Reason}`, `{skip,Reason}`, a `Config` list, or a term describing how *`init_per_suite`* failed.

`NewReturn` is the possibly modified return value of *`init_per_suite`*. To recover from a failure in *`init_per_suite`*, return `ConfigList` with the `tc_status` element removed. For more details, see *Post Hooks* in section "Manipulating Tests" in the User's Guide.

`CTHState` is the current internal state of the CTH.

This function is called only if the CTH is added before or in `init_per_suite`. For details, see section *CTH Scope* in the User's Guide.

```
Module:pre_init_per_group(SuiteName, GroupName, InitData, CTHState) -> Result
```

Types:

```
SuiteName = atom()
GroupName = atom()
InitData = Config | SkipOrFail
Config = NewConfig = [{Key,Value}]
CTHState = NewCTHState = term()
Result = {NewConfig | SkipOrFail, NewCTHState}
SkipOrFail = {fail,Reason} | {skip, Reason}
Key = atom()
Value = term()
Reason = term()
```

OPTIONAL

This function is called before *init_per_group* if it exists. It behaves the same way as *pre_init_per_suite*, but for function *init_per_group* instead.

If `Module:pre_init_per_group/4` is not exported, common_test will attempt to call `Module:pre_init_per_group(GroupName, InitData, CTHState)` instead. This is for backwards compatibility.

```
Module:post_init_per_group(SuiteName, GroupName, Config, Return, CTHState) ->
Result
```

Types:

```
SuiteName = atom()
GroupName = atom()
Config = [{Key,Value}]
Return = NewReturn = Config | SkipOrFail | term()
SkipOrFail = {fail,Reason} | {skip, Reason}
CTHState = NewCTHState = term()
Result = {NewReturn, NewCTHState}
Key = atom()
Value = term()
Reason = term()
```

OPTIONAL

This function is called after *init_per_group* if it exists. It behaves the same way as *post_init_per_suite*, but for function *init_per_group* instead.

If `Module:post_init_per_group/5` is not exported, common_test will attempt to call `Module:post_init_per_group(GroupName, Config, Return, CTHState)` instead. This is for backwards compatibility.

```
Module:pre_init_per_testcase(SuiteName, TestcaseName, InitData, CTHState) ->
Result
```

Types:

```
SuiteName = atom()
TestcaseName = atom()
InitData = Config | SkipOrFail
Config = NewConfig = [{Key,Value}]
CTHState = NewCTHState = term()
Result = {NewConfig | SkipOrFail, NewCTHState}
SkipOrFail = {fail,Reason} | {skip, Reason}
Key = atom()
Value = term()
Reason = term()
```

OPTIONAL

This function is called before *init_per_testcase* if it exists. It behaves the same way as *pre_init_per_suite*, but for function *init_per_testcase* instead.

If `Module:pre_init_per_testcase/4` is not exported, common_test will attempt to call `Module:pre_init_per_testcase(TestcaseName, InitData, CTHState)` instead. This is for backwards compatibility.

CTHs cannot be added here right now. That feature may be added in a later release, but it would right now break backwards compatibility.

`Module:post_init_per_testcase(SuiteName, TestcaseName, Config, Return, CTHState) -> Result`

Types:

```
SuiteName = atom()
TestcaseName = atom()
Config = [{Key,Value}]
Return = NewReturn = Config | SkipOrFail | term()
SkipOrFail = {fail,Reason} | {skip, Reason}
CTHState = NewCTHState = term()
Result = {NewReturn, NewCTHState}
Key = atom()
Value = term()
Reason = term()
```

OPTIONAL

This function is called after *`init_per_testcase`* if it exists. It behaves the same way as *`post_init_per_suite`*, but for function *`init_per_testcase`* instead.

If `Module:post_init_per_testcase/5` is not exported, common_test will attempt to call `Module:post_init_per_testcase(TestcaseName, Config, Return, CTHState)` instead. This is for backwards compatibility.

`Module:pre_end_per_testcase(SuiteName, TestcaseName, EndData, CTHState) -> Result`

Types:

```
SuiteName = atom()
TestcaseName = atom()
EndData = Config
Config = NewConfig = [{Key,Value}]
CTHState = NewCTHState = term()
Result = {NewConfig, NewCTHState}
Key = atom()
Value = term()
Reason = term()
```

OPTIONAL

This function is called before *`end_per_testcase`* if it exists. It behaves the same way as *`pre_end_per_suite`*, but for function *`end_per_testcase`* instead.

This function cannot change the result of the test case by returning skip or fail tuples, but it may insert items in `Config` that can be read in `end_per_testcase/2` or in `post_end_per_testcase/5`.

If `Module:pre_end_per_testcase/4` is not exported, common_test will attempt to call `Module:pre_end_per_testcase(TestcaseName, EndData, CTHState)` instead. This is for backwards compatibility.

```
Module:post_end_per_testcase(SuiteName, TestcaseName, Config, Return,
CTHState) -> Result
```

Types:

`SuiteName = atom()`
`TestcaseName = atom()`
`Config = [{Key,Value}]`
`Return = NewReturn = Config | SkipOrFail | term()`
`SkipOrFail = {fail,Reason} | {skip, Reason}`
`CTHState = NewCTHState = term()`
`Result = {NewReturn, NewCTHState}`
`Key = atom()`
`Value = term()`
`Reason = term()`

OPTIONAL

This function is called after *`end_per_testcase`* if it exists. It behaves the same way as *`post_end_per_suite`*, but for function *`end_per_testcase`* instead.

If `Module:post_end_per_testcase/5` is not exported, common_test will attempt to call `Module:post_end_per_testcase(TestcaseName, Config, Return, CTHState)` instead. This is for backwards compatibility.

```
Module:pre_end_per_group(SuiteName, GroupName, EndData, CTHState) -> Result
```

Types:

`SuiteName = atom()`
`GroupName = atom()`
`EndData = Config | SkipOrFail`
`Config = NewConfig = [{Key,Value}]`
`CTHState = NewCTHState = term()`
`Result = {NewConfig | SkipOrFail, NewCTHState}`
`SkipOrFail = {fail,Reason} | {skip, Reason}`
`Key = atom()`
`Value = term()`
`Reason = term()`

OPTIONAL

This function is called before *`end_per_group`* if it exists. It behaves the same way as *`pre_init_per_suite`*, but for function *`end_per_group`* instead.

If `Module:pre_end_per_group/4` is not exported, common_test will attempt to call `Module:pre_end_per_group(GroupName, EndData, CTHState)` instead. This is for backwards compatibility.

`Module:post_end_per_group(SuiteName, GroupName, Config, Return, CTHState) -> Result`

Types:

```
SuiteName = atom()
GroupName = atom()
Config = [{Key,Value}]
Return = NewReturn = Config | SkipOrFail | term()
SkipOrFail = {fail,Reason} | {skip, Reason}
CTHState = NewCTHState = term()
Result = {NewReturn, NewCTHState}
Key = atom()
Value = term()
Reason = term()
```

OPTIONAL

This function is called after *`end_per_group`* if it exists. It behaves the same way as *`post_init_per_suite`*, but for function *end_per_group* instead.

If `Module:post_end_per_group/5` is not exported, common_test will attempt to call `Module:post_end_per_group(GroupName, Config, Return, CTHState)` instead. This is for backwards compatibility.

`Module:pre_end_per_suite(SuiteName, EndData, CTHState) -> Result`

Types:

```
SuiteName = atom()
EndData = Config | SkipOrFail
Config = NewConfig = [{Key,Value}]
CTHState = NewCTHState = term()
Result = {NewConfig | SkipOrFail, NewCTHState}
SkipOrFail = {fail,Reason} | {skip, Reason}
Key = atom()
Value = term()
Reason = term()
```

OPTIONAL

This function is called before *`end_per_suite`* if it exists. It behaves the same way as *`pre_init_per_suite`*, but for function *`end_per_suite`* instead.

`Module:post_end_per_suite(SuiteName, Config, Return, CTHState) -> Result`

Types:

```
SuiteName = atom()
Config = [{Key,Value}]
Return = NewReturn = Config | SkipOrFail | term()
SkipOrFail = {fail,Reason} | {skip, Reason}
CTHState = NewCTHState = term()
Result = {NewReturn, NewCTHState}
```

Key = atom()
Value = term()
Reason = term()

OPTIONAL

This function is called after *end_per_suite* if it exists. It behaves the same way as *post_init_per_suite*, but for function *end_per_suite* instead.

`Module:on_tc_fail(SuiteName, TestName, Reason, CTHState) -> NewCTHState`

Types:

SuiteName = atom()
TestName = init_per_suite | end_per_suite | {init_per_group,GroupName} | {end_per_group,GroupName} | {FuncName,GroupName} | FuncName
FuncName = atom()
GroupName = atom()
Reason = term()
CTHState = NewCTHState = term()

OPTIONAL

This function is called whenever a test case (or configuration function) fails. It is called after the post function is called for the failed test case, that is:

- If `init_per_suite` fails, this function is called after *post_init_per_suite*.
- If a test case fails, this funcion is called after *post_end_per_testcase*.

If the failed test case belongs to a test case group, the first argument is a tuple `{FuncName,GroupName}`, otherwise only the function name.

The data that comes with `Reason` follows the same format as *FailReason* in event *tc_done*. For details, see section *Event Handling* in the User's Guide.

If `Module:on_tc_fail/4` is not exported, common_test will attempt to call `Module:on_tc_fail(TestName, Reason, CTHState)` instead. This is for backwards compatibility.

`Module:on_tc_skip(SuiteName, TestName, Reason, CTHState) -> NewCTHState`

Types:

SuiteName = atom()
TestName = init_per_suite | end_per_suite | {init_per_group,GroupName} | {end_per_group,GroupName} | {FuncName,GroupName} | FuncName
FuncName = atom()
GroupName = atom()
Reason = {tc_auto_skip | tc_user_skip, term()}
CTHState = NewCTHState = term()

OPTIONAL

This function is called whenever a test case (or configuration function) is skipped. It is called after the post function is called for the skipped test case, that is:

- If `init_per_group` is skipped, this function is called after *post_init_per_group*.
- If a test case is skipped, this function is called after *post_end_per_testcase*.

If the skipped test case belongs to a test case group, the first argument is a tuple `{FuncName,GroupName}`, otherwise only the function name.

The data that comes with `Reason` follows the same format as events *`tc_auto_skip`* and *`tc_user_skip`* For details, see section *Event Handling* in the User's Guide.

If `Module:on_tc_skip/4` is not exported, common_test will attempt to call `Module:on_tc_skip(TestName, Reason, CTHState)` instead. This is for backwards compatibility.

`Module:terminate(CTHState)`

Types:

`CTHState = term()`

OPTIONAL

This function is called at the end of a CTH *scope*.

`Module:id(Opts) -> Id`

Types:

`Opts = term()`

`Id = term()`

OPTIONAL

The `Id` identifies a CTH instance uniquely. If two CTHs return the same `Id`, the second CTH is ignored and subsequent calls to the CTH are only made to the first instance. For details, see section *Installing a CTH* in the User's Guide.

This function is **not** to have any side effects, as it can be called multiple times by `Common Test`.

If not implemented, the CTH acts as if this function returned a call to `make_ref/0`.

ct_property_test

Erlang module

EXPERIMENTAL support in `Common Test` for calling property-based tests.

This module is a first step to run property-based tests in the `Common Test` framework. A property testing tool like QuickCheck or PropEr is assumed to be installed.

The idea is to have a `Common Test` test suite calling a property testing tool with special property test suites as defined by that tool. The usual Erlang application directory structure is assumed. The tests are collected in the `test` directory of the application. The `test` directory has a subdirectory `property_test`, where everything needed for the property tests is collected.

A typical `Common Test` test suite using `ct_property_test` is organized as follows:

```
-include_lib("common_test/include/ct.hrl").

all() -> [prop_ftp_case].

init_per_suite(Config) ->
    ct_property_test:init_per_suite(Config).

%%%---- test case
prop_ftp_case(Config) ->
    ct_property_test:quickcheck(
      ftp_simple_client_server:prop_ftp(Config),
      Config
     ).
```

Warning:

This is experimental code that can be changed or removed anytime without any warning.

Exports

`init_per_suite(Config) -> Config | {skip, Reason}`

Initializes `Config` for property testing.

This function investigates if support is available for either Quickcheck, PropEr, or Triq. The options `{property_dir,AbsPath}` and `{property_test_tool,Tool}` are set in the `Config` returned.

The function is intended to be called in function `init_per_suite` in the test suite.

The property tests are assumed to be in subdirectory `property_test`.

`quickcheck(Property, Config) -> true | {fail, Reason}`

Calls quickcheck and returns the result in a form suitable for `Common Test`.

This function is intended to be called in the test cases in the test suite.

ct_testspec

Erlang module

Parsing of test specifications for `Common Test`.

This module exports help functions for parsing of test specifications.

Exports

`get_tests(SpecsIn) -> {ok, [{Specs,Tests}]} | {error, Reason}`

Types:

```
SpecsIn = [string()] | [[string()]]
Specs = [string()]
Test = [{Node,Run,Skip}]
Node = atom()
Run = {Dir,Suites,Cases}
Skip = {Dir,Suites,Comment} | {Dir,Suites,Cases,Comment}
Dir = string()
Suites = atom | [atom()] | all
Cases = atom | [atom()] | all
Comment = string()
Reason = term()
```

Parse the given test specification files and return the tests to run and skip.

If `SpecsIn=[Spec1,Spec2,...]`, separate tests will be created per specification. If `SpecsIn=[[Spec1,Spec2,...]]`, all specifications will be merge into one test.

For each test, a `{Specs,Tests}` element is returned, where `Specs` is a list of all included test specifications, and `Tests` specifies actual tests to run/skip per node.