

Common Test

Copyright © 2003-2017 Ericsson AB. All Rights Reserved.
Common Test 1.10.1
February 22, 2017

Copyright © 2003-2017 Ericsson AB. All Rights Reserved.
The contents of this file are subject to the Erlang Public License, Version 1.1, (the "License"); you may not use this file except in compliance with the License. You should have received a copy of the Erlang Public License along with this software. If not, it can be retrieved online at http://www.erlang.org/. Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License. Ericsson AB. All Rights Reserved..

February 22, 2017

1 Common Test User's Guide

Common Test is a portable application for automated testing. It is suitable for black-box testing of target systems of any type (i.e. not necessarily implemented in Erlang), as well as for white-box testing of Erlang/OTP programs. Black-box testing is performed via standard O&M interfaces (such as SNMP, HTTP, Corba, Telnet, etc) and, if required, via user specific interfaces (often called test ports). White-box testing of Erlang/OTP programs is easily accomplished by calling the target API functions directly from the test case functions. Common Test also integrates usage of the OTP cover tool for code coverage analysis of Erlang/OTP programs.

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

Common Test provides libraries that contain useful support functions to fill various testing needs and requirements. There is for example support for flexible test declarations by means of so called test specifications. There is also support for central configuration and control of multiple independent test sessions (towards different target systems) running in parallel.

Common Test is implemented as a framework based on the OTP Test Server application.

1.1 Common Test Basics

1.1.1 Introduction

The *Common Test* framework (CT) is a tool which supports implementation and automated execution of test cases towards arbitrary types of target systems. The CT framework is based on the OTP Test Server and it's 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 (a feature that makes it possible to test multiple systems independently in one common session, useful e.g. for running automated large-scale regression tests).

The SUT (System Under Test) may consist of one or several target nodes. CT contains a generic test server which, together with other test utilities, is used to perform test case execution. It is possible to start the tests from a GUI or from the OS- or 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 utilize in order to carry out the tests.

In a black-box testing scenario, CT based test programs connect to the target system(s) via standard O&M and CLI protocols. CT provides implementations of, and wrapper interfaces to, some of these protocols (most of which exist as stand-alone components and applications in OTP). The wrappers simplify configuration and add verbosity for logging purposes. CT will be continously extended with useful support modules. (Note however that it's a straightforward task to use any arbitrary Erlang/OTP component for testing purposes with Common Test, without needing a CT wrapper for it. It's as simple as calling Erlang functions). There are a number of target independent interfaces supported in CT, such as Generic Telnet, FTP, etc, which can be specialized or used directly for controlling instruments, traffic load generators, etc.

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

A test case can handle several connections towards one or several target systems, instruments and traffic generators in parallel in order 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 CT users can take great advantage of!).

1.1.2 Test Suite Organisation

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

1.1.3 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 CT framework, and the various libraries and applications provided by Erlang/OTP, there might also be a need for customized (user specific) support libraries.

1.1.4 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 which tests one or more things. The test case is the smallest unit that the CT test server deals with.

Subsets of test cases, called test case groups, may also be defined. A test case group can have execution properties associated with it. Execution properties specify whether the test cases in the group should be executed in random order, in parallel, in sequence, and if the execution of the group should be repeated. Test case groups may also be nested (i.e. a group may, besides test cases, contain sub-groups).

Besides test cases and groups, the test suite may also contain configuration functions. These functions are meant to be used for setting up (and verifying) environment and state on the SUT (and/or the CT host node), required for the tests to execute correctly. Examples of operations: Opening a connection to the SUT, initializing a database, running an installation script, etc. Configuration may 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 CT test server. See the *Writing Test Suites* chapter for more information.

A test case is considered successful if it returns to the caller, no matter what the returned value is. A few return values have special meaning however (such as `{skip,Reason}` which indicates that the test case is skipped, `{comment,Comment}` which prints a comment in the log for the test case and `{save_config,Config}` which makes the CT 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 will be concise and readable test case functions that look much more like scripts than actual programs. 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 several 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 row in a table showing total execution time, whether 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.

1.1.5 External Interfaces

The CT 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()
: Info 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)
: Info 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 CT test server expects these functions:

Testcasename()
: Info function that returns a list of test case properties. (Optional)

Testcasename(Config)
: The actual test case function.

1.2 Getting Started

1.2.1 Are you new around here?

The purpose of this short chapter is to, with a "learning by example" approach, give the newcomer a chance to get started quickly writing and executing some first simple tests. The chapter will introduce some of the basics, but leave most explanations and details for the later chapters in this User's Guide. Hopefully though, after this chapter, you will be inspired and unintimidated enough to go on and get into the nitty-gritty that follows in this rather heavy User's Guide! If you're not much into "learning by example" and prefer to get into more technical detail right away, go ahead and skip to the next chapter. Again, the basics presented here will be covered in detail in later chapters.

This chapter also tries to demonstrate how dead simple it actually is to write a very 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 rest of the chapters in the User's Guide.

A quick note before we start: In order to understand what's discussed and examplified here, it is recommended that you first read through the opening *Common Test Basics* chapter.

1.2.2 Test case execution

Execution of test cases is handled this way:

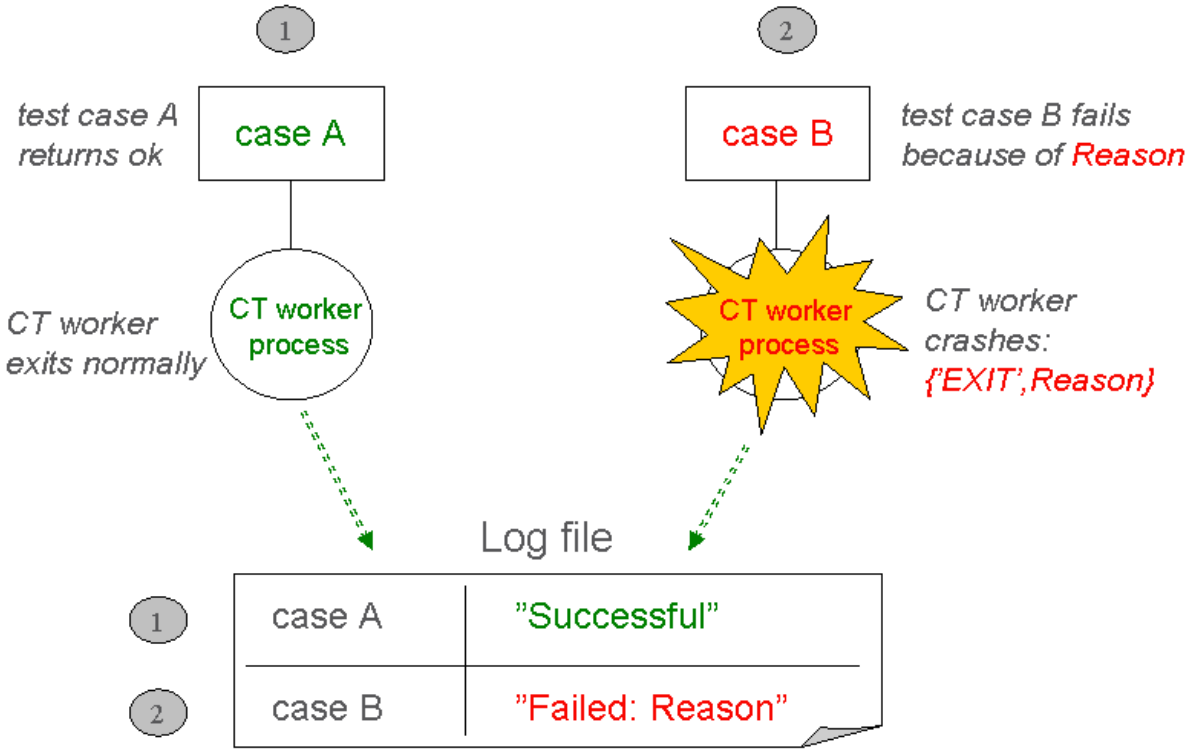


Figure 2.1: Successful vs unsuccessful test case execution.

For each test case that Common Test is told to execute, it spawns a dedicated process on which the test case function in question 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 and this is what's 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 simply returns a value and Common Test logs the test case as successful.

In scenario 2, an error is detected during test case execution which causes the test case B function to generate an exception. This causes the test case process to exit with reason other than normal, and as a result, Common Test will log this as an unsuccessful test case.

As you can understand from the illustration above, Common Test requires that a test case generates a runtime error to indicate failure (e.g. by causing a bad match error or by calling `exit/1`, preferrably through the `ct:fail/1,2` help function). A succesful execution is indicated by means of a normal return from the test case function.

1.2.3 A simple test suite

As you've seen in the basics chapter, the test suite module implements *callback functions* (mandatory or optional) for various purposes, e.g:

- 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 and if you don't need them for your test, a test suite with one simple test case could look like this:

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

all() ->
    [mod_exists].

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

In this example we check that the `mymod` module exists (i.e. can be successfully loaded by the code server). If the operation fails, we will get a bad match error which terminates the test case.

1.2.4 A test suite with configuration functions

If we need to perform configuration operations in order to run our test, we implement configuration functions in our suite. The result from a configuration function is configuration data, or simply *`Config`*. This is a list of key-value tuples which get passed from the configuration function to the test cases (possibly through configuration functions on "lower level"). The data flow looks like this:

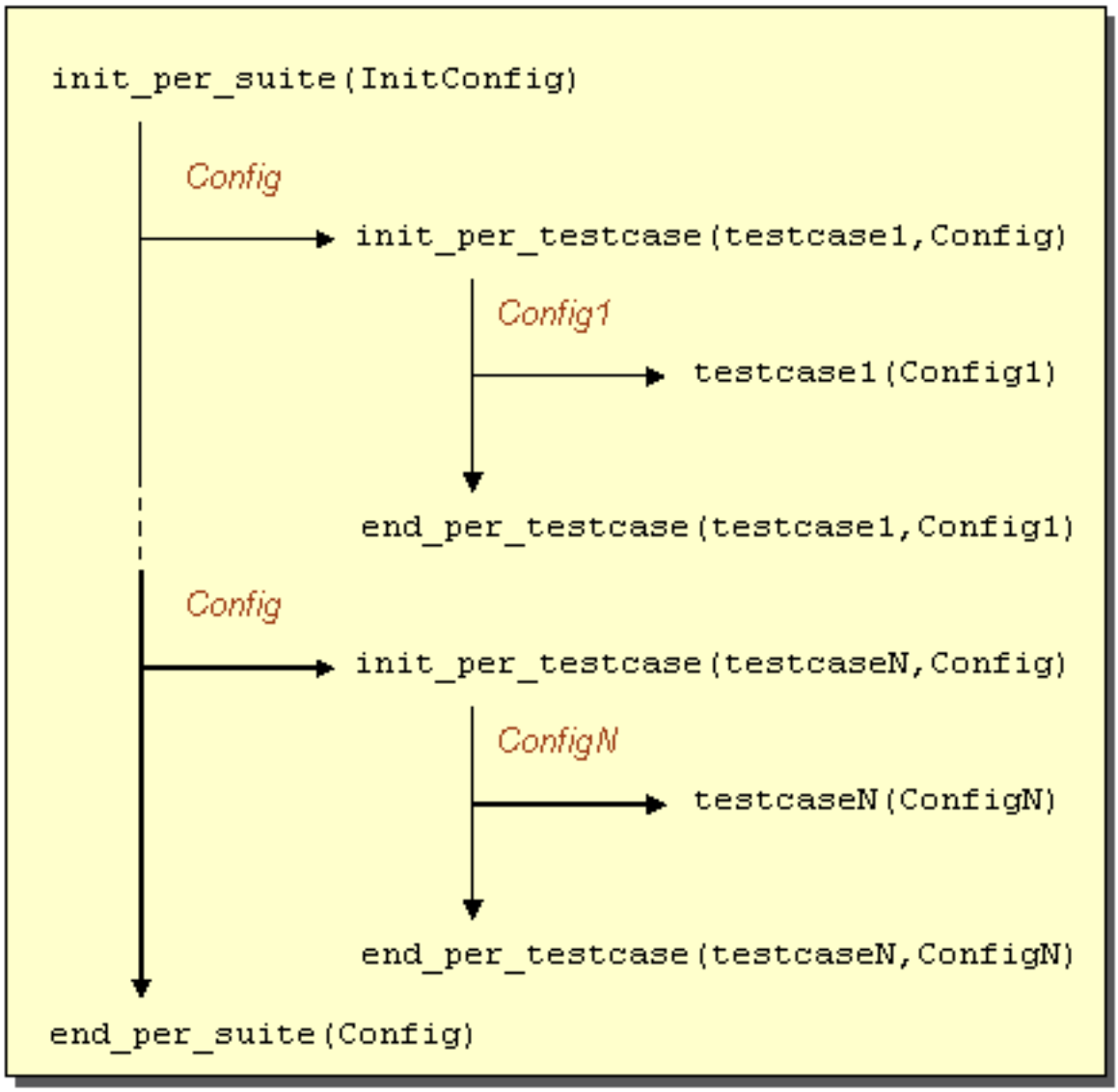


Figure 2.2: Config data flow in the suite.

Here's an example of a test suite which uses configuration functions to open and close a log file for the test cases (an operation that would be 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.
```

In this example we have test cases that verify, by parsing a log file, that our SUT has performed a successful restart and that no unexpected errors have been printed.

To execute the test cases in the test suite above, we could type this on the Unix/Linux command line (assuming for this example that the suite module is in the current working directory):

```
$ ct_run -dir .
```

or

```
$ ct_run -suite check_log_SUITE
```

If we want to use the Erlang shell to run our test, we could evaluate this call:

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

or

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

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

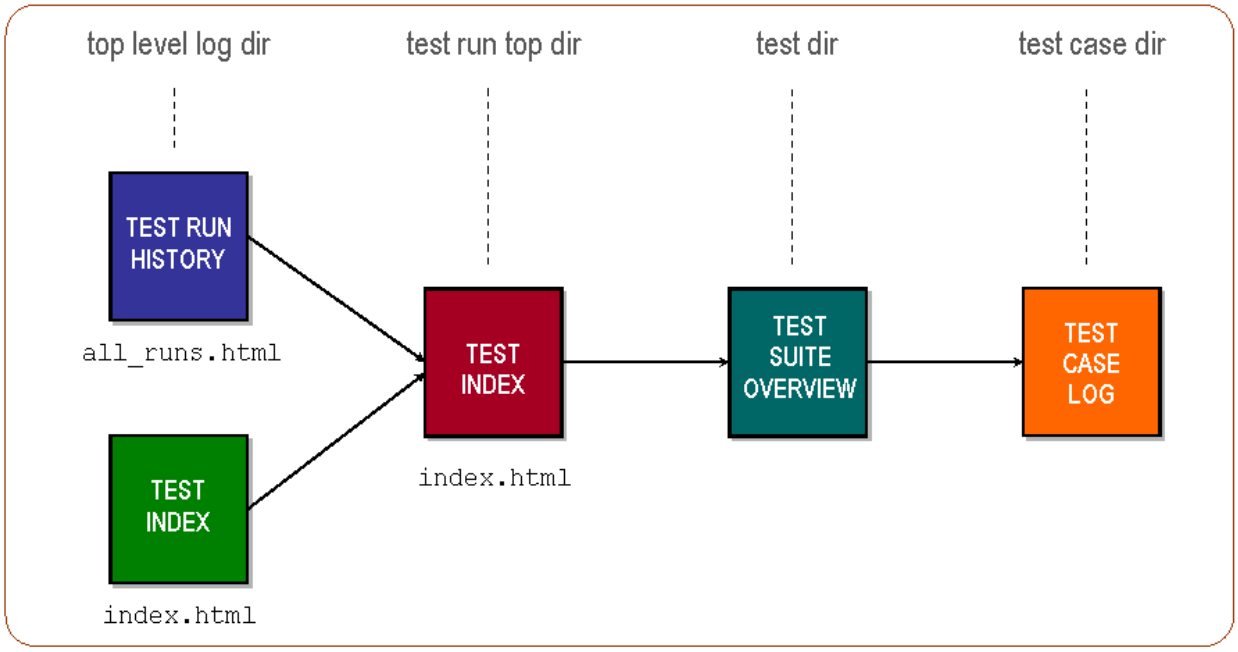


Figure 2.3: HTML log file structure.

1.2.5 What happens next?

Well, you might already be asking yourself questions such as:

- "How and where can I specify variable data for my tests that mustn't be hard-coded in the test suites (such as host names, addresses, user login data, etc)?" The *External Configuration Data* chapter will give you that information.
- "Is there a way to declare a number of different tests and run them in one session without having to write my own scripts? And can such declarations be used for regression testing?" The *Test Specifications* chapter answers these questions.
- "Can test cases and/or test runs be automatically repeated?" Learn more about *Test Case Groups* and also read about start flags/options in the *Running Tests* chapter and the Reference Manual.
- "Will Common Test execute my test cases in sequence or in parallel?" The *Test Case Groups* section in the Running Tests chapter will give you the answer.
- "What's the syntax for timetraps (mentioned above), and how do I set them?" This is explained in the *Timetrap Timeouts* part of the Writing Test Suites chapter.
- "What functions are available for logging and printing?" Check the *Logging* section in the Writing Test Suites chapter.
- "I need data files for my tests. Where do I store them preferrably?" You should read about *Data and Private Directories* for information about this.
- "May I start with a test suite example, please?" *Sure!*

You will 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. You will find that there's lots more to learn about the things that have been introduced in this chapter. You will of course also be presented many more useful features, such as the ones listed above. Have fun!

1.3 Installation

1.3.1 General information

The two main interfaces for running tests with Common Test are an executable program named ct_run and an erlang module named `ct`. The ct_run program is compiled for the underlying operating system (e.g. 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.

A legacy Bourne shell script - named run_test - exists, which may be manually generated and installed. This script may be used instead of the ct_run program mentioned above, e.g. if the user wishes to modify or customize the Common Test start flags in a simpler way than making changes to the ct_run C program.

The Common Test application is installed with the Erlang/OTP system and no additional installation step is required to start using Common Test by means of the ct_run executable program, and/or the interface functions in the `ct` module. If you wish to use the legacy Bourne shell script version run_test, however, this script needs to be generated first, according to the instructions below.

> **Note:**
>
> Before reading on, please note that since Common Test version 1.5, the run_test shell script is no longer required for starting tests with Common Test from the OS command line. The ct_run program (descibed above) is the new recommended command line interface for Common Test. The shell script exists mainly for legacy reasons and may not be updated in future releases of Common Test. It may even be removed.

Optional step to generate a shell script for starting Common Test:

To generate the run_test shell script, navigate to the `common_test-<vsn>` directory, located among the other OTP applications (under the OTP lib directory). Here execute the `install.sh` script with argument `local`:

```
$ ./install.sh local
```

This generates the executable run_test script in the `common_test-<vsn>/priv/bin` directory. The script will include absolute paths to the Common Test and Test Server application directories, so it's possible to copy or move the script to a different location on the file system, if desired, without having to update it. It's of course possible to leave the script under the `priv/bin` directory and update the PATH variable accordingly (or create a link or alias to it).

If you, for any reason, have copied Common Test and Test Server to a different location than the default OTP lib directory, you can generate a run_test script with a different top level directory, simply by specifying the directory, instead of `local`, when running `install.sh`. Example:

```
$ install.sh /usr/local/test_tools
```

Note that the `common_test-<vsn>` and `test_server-<vsn>` directories must be located under the same top directory. Note also that the install script does not copy files or update environment variables. It only generates the run_test script.

Whenever you install a new version of Erlang/OTP, the run_test script needs to be regenerated, or updated manually with new directory names (new version numbers), for it to "see" the latest Common Test and Test Server versions.

1.4 Writing Test Suites

1.4.1 Support for test suite authors

The `ct` module provides the main interface for writing test cases. This includes e.g:

- 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

Please see the reference manual for the `ct` module for details about these functions.

The CT application also includes other modules named `ct_<component>` that provide various support, mainly simplified use of communication protocols such as rpc, snmp, ftp, telnet, etc.

1.4.2 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 CT will not be able to locate it (at least not per default).

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

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

The callback functions that the test suite should implement, and which will be described in more detail below, are all listed in the *common_test reference manual page*.

1.4.3 Init and end per suite

Each test suite module may 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 it exists, `init_per_suite` is called initially before the test cases are executed. It typically contains initializations that are common for all test cases in the suite, and that are only to be performed once. It is recommended to be used for setting up and verifying state and environment on the SUT (System Under Test) and/or the CT host node, so that the test cases in the suite will execute correctly. Examples of initial configuration operations: Opening a connection to the SUT, initializing a database, running an installation script, etc.

`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` will 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`, 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 will be skipped automatically (so called *auto skipped*), including `end_per_suite`.

Note 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 may be saved to the log files for these dummies (see the *Common Test Hooks* chapter for more information).

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

`end_per_testcase/2` is called after each test case has finished, giving the opportunity to perform clean-up 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 a number of, 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` may modify this parameter or return it as is. The return value of `init_per_testcase/2` is passed as the `Config` parameter 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.

It is possible in `end_per_testcase` to check if the test case was successful or not (which consequently may determine how cleanup should be performed). This is done by reading the value tagged with `tc_status` from `Config`. The value is either `ok`, `{failed,Reason}` (where `Reason` is `timetrap_timeout`, info from `exit/1`, or details of a run-time error), or `{skipped,Reason}` (where Reason is a user specific term).

The `end_per_testcase/2` function is called even after a test case terminates due to a call to *`ct:abort_current_testcase/1`*, or after a timetrap timeout. However, `end_per_testcase` will then execute on a different process than the test case function, and in this situation, `end_per_testcase` will not be able to change the reason for test case termination by returning `{fail,Reason}`, nor will it be able to save data with `{save_config,Data}`.

If `init_per_testcase` crashes, the test case itself gets skipped automatically (so called *auto skipped*). If `init_per_testcase` returns a tuple `{skip,Reason}`, also then the test case gets skipped (so called *user skipped*). It is also possible, by returning a tuple `{fail,Reason}` from `init_per_testcase`, to mark the test case as failed without actually executing it.

> **Note:**
>
> If `init_per_testcase` crashes, or returns `{skip,Reason}` or `{fail,Reason}`, the `end_per_testcase` function is not called.

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

`init_per_testcase` and `end_per_testcase` execute on the same Erlang process as the test case and printouts from these configuration functions can be found in the test case log file.

1.4.5 Test cases

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

It is possible to choose to put many or few tests into each test case. What exactly each test case does is of course up to the author, but here are some things to keep in mind:

Having 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. Duplicated code should be avoided, e.g. by means of common help functions, or the resulting suite will be difficult to read and understand, and expensive to maintain.

Larger test cases make it harder to tell what went wrong if it fails, and large portions of test code will potentially be skipped when errors occur. Furthermore, readability and maintainability suffers when test cases become too large and extensive. Also, the resulting log files may not reflect very well the number of tests that have actually been performed.

The test case function takes one argument, `Config`, which contains configuration information such as `data_dir` and `priv_dir`. (See *Data and Private Directories* for more information about these). The value of `Config` at the time of the call, is the same as the return value from `init_per_testcase`, see above.

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

Since the `Config` parameter is a list of key-value tuples, i.e. a data type generally called a property list, it can be handled by means of the `proplists` module in the OTP `stdlib`. A value can for example be searched for and returned with the `proplists:get_value/2` function. Also, or alternatively, you might want to look in the general `lists` module, also in `stdlib`, for useful functions. Normally, the only operations you ever perform 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 actual 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 gets logged as such.

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

1.4.6 Test case info function

For each test case function there can be an additional function with the same name but with no arguments. This is the test case info function. The test case info function is expected to return a list of tagged tuples that specifies various properties regarding the test case.

The following tags have special meaning:

`timetrap`

Set the maximum time the test case is allowed to execute. If the timetrap time is exceeded, the test case fails with reason `timetrap_timeout`. Note that `init_per_testcase` and `end_per_testcase` are included in the timetrap time. Please see the *Timetrap* section for more details.

`userdata`

Use this to specify arbitrary data related to the testcase. This data can be retrieved at any time using the *`ct:userdata/3`* utility function.

`silent_connections`

Please see the *Silent Connections* chapter for details.

require

Use this to specify configuration variables that are 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.

It is also possible to give a required variable a default value that will 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 info 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"}]}}].
```

See the *Config files* chapter and the *ct:require/1/2* function in the *ct* reference manual for more information about `require`.

> **Note:**
>
> Specifying a default value for a required variable can result in a test case always getting executed. This might not be a desired behaviour!

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

Other tags than the ones mentioned above will simply be ignored by the test server.

Example of a test case info function:

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

1.4.7 Test suite info function

The `suite/0` function can be used in a test suite module to e.g. set a default `timetrap` value and to `require` external configuration data. If a test case-, or group info function also specifies any of the info tags, it overrides the default values set by `suite/0`. See the test case info function above, and group info function below, for more details.

Other options that may be specified with the suite info list are:

- `stylesheet`, see *HTML Style Sheets*.
- `userdata`, see *Test case info function*.
- `silent_connections`, see *Silent Connections*.

Example of the suite info function:

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

1.4.8 Test case groups

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

```
     groups() -> GroupDefs

     Types:

     GroupDefs = [GroupDef]
     GroupDef = {GroupName,Properties,GroupsAndTestCases}
     GroupName = atom()
     GroupsAndTestCases = [GroupDef | {group,GroupName} | TestCase]
     TestCase = atom()
```

`GroupName` is the name of the group and should be unique within the test suite module. Groups may be nested, and this is accomplished simply 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:

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

If the `parallel` property is specified, Common Test will execute all test cases in the group in parallel. If `sequence` is specified, the cases will be executed in a sequence, as described in the chapter *Dependencies between test cases and suites*. If `shuffle` is specified, the cases in the group will be executed in random order. The `repeat` property orders Common Test to repeat execution of 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 should be executed (also with respect to test cases that are not part of any group), tuples on the form `{group,GroupName}` should be added to the `all/0` list. Example:

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

It is also possible to specify execution properties with a group tuple in `all/0`: `{group,GroupName,Properties}`. These properties will override those specified in the group definition (see `groups/0` above). This way, it's possible to run the same set of tests, but with different properties, without having to make copies of the group definition in question.

If a group contains sub-groups, the execution properties for these may 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 sub-groups. Any sub-groups defined in `group/0` for a group, that are not specified in the `SubGroups` list, will simply execute with their pre-defined 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}]}]}].
```

Note that this is equivalent to this specification:

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

The value `default` states that the pre-defined properties should be used.

Here's an example of 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 syntax described above may also be used in Test Specifications in order to change properties of groups at the time of execution, without even having to edit the test suite (please see the *Test Specifications* chapter for more info).

As illustrated above, properties may be combined. If e.g. `shuffle`, `repeat_until_any_fail` and `sequence` are all specified, the test cases in the group will be executed repeatedly, and in random order, until a test case fails. Then execution is immediately stopped and the rest of the cases 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 means of the `Config` argument. `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, the `end_per_group(GroupName, Config` function is called. This function is meant to be used for cleaning up after `init_per_group/2`.

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 will be saved to the log files for these dummies (see the *Common Test Hooks* chapter for more information).

> 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 is always printed on the top of the HTML log for `init_per_group/2`. Also, the total execution time for a group can be found at the bottom of the log for `end_per_group/2`.

Test case groups may be nested so that sets of groups can be configured with the same `init_per_group/2` and `end_per_group/2` functions. Nested groups may 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 example above, if `all/0` would return group name references in this order: `[{group,group1}, {group,group3}]`, the order of the configuration functions and test cases will be the following (note 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 not actually
        defined since 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 sub-groups. E.g, in the example above, the test cases in `group2` will not be executed in random order (which is the property of `group1`).

1.4.9 The parallel property and nested groups

If a group has a parallel property, its test cases will be spawned simultaneously and get executed in parallel. A test case is not allowed to execute in parallel with `end_per_group/2` however, which means that the time it takes 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 the `end_per_group/2` function for the group has finished.

A group nested under a parallel group will start executing in parallel with previous (parallel) test cases (no matter what properties the nested group has). Since, however, test cases are never executed in parallel with `init_per_group/2` or `end_per_group/2` of the same group, it's only after a nested group has finished that any remaining parallel cases in the previous group get spawned.

1.4.10 Parallel test cases and IO

A parallel test case has a private IO server as its group leader. (Please see the Erlang Run-Time System Application documentation for a description of the group leader concept). The central IO server process that handles the output from regular test cases and configuration functions, does not respond to IO messages during execution of parallel groups. This is important to understand in order to avoid certain traps, like this one:

If a process, `P`, is spawned during execution of e.g. `init_per_suite/1`, it will inherit the group leader of the `init_per_suite` process. This group leader is the central IO server process mentioned above. If, at a later time, *during parallel test case execution*, some event triggers process `P` to call `io:format/1/2`, that call will never return (since the group leader is in a non-responsive state) and cause `P` to hang.

1.4.11 Repeated groups

A test case group may be repeated a certain number of times (specified by an integer) or indefinitely (specified by `forever`). The repetition may also be stopped prematurely if any or all cases fail or succeed, i.e. if the property `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.

It is possible to return the status of a sub-group (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` should be returned from the group as a result, `end_per_group/2` should return the value `{return_group_result,failed}`. The status of a sub-group is taken into account by Common Test when evaluating if execution of a group should be repeated or not (unless the basic `repeat` property is used).

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

Here's 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 sub-group (maybe to determine what status the current group should also return). This is as simple as illustrated in the example above, only the name of the group 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.

1.4.12 Shuffled test case order

The order that 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 the `shuffle` property set, however, Common Test will instead execute the test cases in random order.

The user may 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 given, Common

Test creates a "random" seed for the shuffling operation (using the return value of `erlang:now()`). 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 will not be reset in between turns.

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

1.4.13 Group info function

The test case group info function, `group(GroupName)`, serves the same purpose as the suite- and test case info functions previously described in this chapter. The scope for the group info, however, is all test cases and sub-groups in the group in question (`GroupName`).

Example:

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

The group info properties override those set with the suite info function, and may in turn be overridden by test case info properties. Please see the test case info function above for a list of valid info properties and more general information.

1.4.14 Info functions for init- and end-configuration

It is possible to use info functions also for the `init_per_suite`, `end_per_suite`, `init_per_group`, and `end_per_group` functions, and it works the same way as with info functions for test cases (see above). This is useful e.g. 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 info function `init/end_per_suite()` is called for `init/end_per_suite(Config)`, and info function `init/end_per_group(GroupName)` is called for `init/end_per_group(GroupName,Config)`. Info functions can not be used with `init/end_per_testcase(TestCase, Config)`, however, since these configuration functions execute on the test case process and will use the same properties as the test case (i.e. the properties set by the test case info function, `TestCase()`). Please see the test case info function above for a list of valid info properties and more general information.

1.4.15 Data and Private Directories

The data directory, `data_dir`, is the directory where the test module has its own files needed for the testing. The name of the `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, i.e. to avoid hardcoding directory names in your suite. Since the data directory is stored in the same directory as your test suite, you should be able to 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 may 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 that all test cases share. This may not always be suitable, especially if the same test cases are executed multiple times during a test run (e.g. if they belong to a test case group with repeat property), and there's a risk that files in the private directory get overwritten. Under these circumstances, it's possible to configure Common Test to create one dedicated private directory per test case and execution instead. This is accomplished by means of 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:

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

The first value indicates the default priv_dir behaviour, i.e. 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 will be created automatically. This can obviously become very inefficient for test runs with many test cases and/or repetitions. Therefore, in case the manual version is instead used, 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:**
>
> You should not depend on current working directory for reading and writing data files since this is not portable. All scratch files are to be written in the `priv_dir` and all data files should be located in `data_dir`. Note also that the Common Test server sets current working directory to the test case log directory at the start of every case.

1.4.16 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.

1.4.17 Timetrap timeouts

The default time limit for a test case is 30 minutes, unless a `timetrap` is specified either by the suite-, group-, or test case info function. The timetrap timeout value defined by `suite/0` is the value that will be used for each test case in the suite (as well as 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 will be used for each test case in group `GroupName`, and any of its sub-groups. If a timetrap value is defined by `group/1` for a sub-group, it overrides that of its higher level groups. Timetrap values set by individual test cases (by means of the test case info function) override both group- and suite- level timetraps.

It is also possible to dynamically set/reset a timetrap during the excution 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 timeout, or end of the current function).

Timetrap values can be extended with a multiplier value specified at startup with the `multiply_timetraps` option. It is also possible to let the test server decide to scale up timetrap timeout values automatically, e.g. if tools

such as cover or trace are running during the test. This feature is disabled by default and can be enabled with the `scale_timetraps` start option.

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`* may be used (instead of e.g. `timer:sleep/1`).

A function (`fun/0` or `MFA`) may be specified as timetrap value in the suite-, group- and test case info function, as well as argument to the *`ct:timetrap/1`* function. Examples:

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

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

The user timetrap function may be used for two things:

- To act as a timetrap - the timeout 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 timeout is triggered, *unless* the return value is a valid timetrap time, such as an integer, or a `{SecMinOrHourTag,Time}` tuple (see the *common_test reference manual* for details). If a time value is returned, a new timetrap is started to generate a timeout after the specified time.

The user timetrap function may of course return a time value after a delay, and if so, the effective timetrap time is the delay time *plus* the returned time.

1.4.18 Logging - categories and verbosity levels

Common Test provides three main functions for printing strings:

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

The `log/1/2/3/4` function will print a string to the test case log file. The `print/1/2/3/4` function will print the string to screen, and the `pal/1/2/3/4` function will print the same string both to file and screen. (The functions are documented in the `ct` reference manual).

The optional `Category` argument may be used to categorize the log printout, and categories can be used for two things:

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

The `Importance` argument specifies a level of importance which, compared to a verbosity level (general and/or set per category), determines if the printout should be visible or not. `Importance` is an arbitrary integer in the range 0..99. Pre-defined constants exist in the `ct.hrl` header file. The default importance level, `?STD_IMPORTANCE` (used if the `Importance` argument is not provided), is 50. This is also the importance used for standard IO, e.g. from printouts made with `io:format/2`, `io:put_chars/1`, etc.

`Importance` is compared to a verbosity level set by means of the `verbosity` start flag/option. The verbosity level can be set per category and/or generally. The default verbosity level, `?STD_VERBOSITY`, is 50, i.e. all standard IO gets printed. If a lower verbosity level is set, standard IO printouts will be ignored. Common Test performs the following test:

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

This also means that verbosity level 0 effectively turns all logging off (with the exception of 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 IO printouts, uncategorized `ct:log/print/pal` printouts, as well as printouts for categories with undefined verbosity level.

Example:

```
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, "5. Categorized error, importance = ~w", [?HI_IMPORTANCE]),
  ct:log(error, ?HI_IMPORTANCE, "6. Categorized error, importance = ~w", [?MAX_IMPORTANCE]),

If starting the test without specifying any verbosity levels:

  $ ct_run ...

the following gets 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 gets printed:

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

How categories can be mapped to CSS tags is documented in the *Running Tests* chapter.

The `Format` and `Args` arguments in `ct:log/print/pal` are always passed on to the `io:format/3` function in `stdlib` (please see the `io` manual page for details).

For more information about log files, please see the *Running Tests* chapter.

1.4.19 Illegal dependencies

Even though it is highly efficient to write test suites with the Common Test framework, there will surely be mistakes made, mainly due to illegal dependencies. Noted below are some of the more frequent mistakes from our own experience with running the Erlang/OTP test suites.

- 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 what 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` should be used to locate data and for writing scratch files.
- Depending on execution order:

During development of test suites, no assumption should preferrably be made about the execution order of the test cases or suites. E.g. a test case should not assume that a server it depends on, has already been started by a previous test case. There are several reasons for this:

Firstly, the user/operator may specify the order at will, and maybe a different execution order is more relevant or efficient on some particular occasion. Secondly, if the user specifies a whole directory of test suites for his/her test, the order the suites are executed will depend on how the files are listed by the operating system, which varies between systems. Thirdly, if a user wishes 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:

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

  Functionality common for many test case functions may 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 may be ok if the test case will fail at a later stage, but it is never acceptable just to print an error message (into the log file) and return successfully. Such test cases do harm since they create a false sense of security when overviewing the test results.

- Messing up for subsequent test cases:

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

1.5 Test Structure

1.5.1 Test structure

A test is performed by running one or more test suites. A test suite consists of test cases (as well as configuration functions and info functions). Test cases may 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.

1.5.2 Skipping test cases

It is possible to skip certain test cases, for example if you know beforehand that a specific test case fails. This might be functionality which isn't yet implemented, a bug that is known but not yet fixed or some functionality which doesn't work or isn't applicable on a specific platform.

There are several different ways to state that one or more test cases should be skipped:

- Using `skip_suites` and `skip_cases` terms in *test specifications*.
- Returning `{skip,Reason}` from the `init_per_testcase/2` or `init_per_suite/1` functions.
- Returning `{skip,Reason}` from the execution clause of the test case.

The latter of course means that the execution clause is actually called, so the author must make sure that the test case does not run.

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

1.5.3 Definition of terms

Auto skipped test case
: When a configuration function fails (i.e. terminates unexpectedly), the test cases that depend on the configuration function will be 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 required configuration data is not available 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 SUT (System Under Test) 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 that contains data related to a test and/or an SUT (System Under Test), e.g. protocol server addresses, client login details, hardware interface addresses, etc - any data that should be handled as variable in the suite and not be hardcoded.

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, e.g. additional Erlang modules, binaries or data files.

Info 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 should be used when the test suite needs to write to files.

ct_run
: The name of an executable program that may 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 that share configuration functions and execution properties. The execution properties specify whether the test cases in the group should be executed in random order, in parallel, in sequence, and if the execution of the group should be repeated. Test case groups may also be nested (i.e. a group may, besides test cases, contain sub-groups).

Test suite
: An erlang module containing a collection of test cases for a specific functional area.

Test directory
: A directory that contains one or more test suite modules, i.e. a group of test suites.

The Config argument
: A list of key-value tuples (i.e. a property list) containing runtime configuration data passed from the configuration functions to the test cases.

User skipped test case
: This is the status of a test case that has been explicitly skipped in any of the ways described in the "Skipping test cases" section above.

1.6 Examples and Templates

1.6.1 Test suite example

This 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) -> void()
%%
%% 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) -> void()
%%
%% 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.
```

1.6.2 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 you a good overview of the available callback functions. Here are the templates in question:

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) -> void() | {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) ->
%%               void() | {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) ->
%%               void() | {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) -> void() | {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) ->
%%               void() | {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) ->
%%               void() | {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.
```

1.7 Running Tests and Analyzing Results

1.7.1 Using the Common Test Framework

The Common Test Framework provides a high level operator interface for testing. It adds the following features to the Erlang/OTP Test Server:

- Automatic compilation of test suites (and help modules).
- Creation of additional HTML pages for better overview.
- Single command interface for running all available tests.
- Handling of configuration files specifying data related to the System Under Test (and any other variable data).
- Mode for running multiple independent test sessions in parallel with central control and configuration.

1.7.2 Automatic compilation of test suites and help modules

When Common Test starts, it will automatically attempt to compile any suites included in the specified tests. If particular suites have been specified, only those suites will be compiled. If a particular test object directory has been specified (meaning all suites in this directory should be part of the test), Common Test runs make:all/1 in the directory to compile the suites.

If compilation should fail for one or more suites, the compilation errors are printed to tty and the operator is asked if the test run should proceed without the missing suites, or be aborted. If the operator chooses to proceed, it is noted in the HTML log which tests have missing suites. If Common Test is unable to prompt the user after compilation failure (if Common Test doesn't control stdin), the test run will proceed automatically without the missing suites. This behaviour 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 will stop immediately if some suites fail to compile.

Any help module (i.e. 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, will also be automatically compiled. A help module will not be mistaken for a test suite (unless it has a "_SUITE" name of course). 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, you may specify these include directories by means of the `-include` flag with `ct_run`, or the `include` option with `ct:run_test/1`.

In addition to this, an include path may 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 will pass all include directories (specified either with the `include` flag/option, or the `CT_INCLUDE_PATH` variable, or both) to the compiler.

It is also possible to specify include directories in test specifications (see below).

If the user wants to run all test suites for a test object (or OTP application) by specifying only the top directory (e.g. with the `dir` start flag/option), Common Test will primarily look for test suite modules in a subdirectory named `test`. If this subdirectory doesn't exist, the specified top directory is assumed to be the actual test directory, and test suites will be read from there instead.

It is possible to disable the automatic compilation feature by using the `-no_auto_compile` flag with `ct_run`, or the `{auto_compile,false}` option 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 can not be loaded from the local file system during startup of Common Test, the user needs to pre-load the modules before starting the test. Common Test will only verify that the specified test suites exist (i.e. that they are, or can be, loaded). This is useful e.g. if the test suites are transferred and loaded as binaries via RPC from a remote node.

1.7.3 Running tests from the OS command line

The `ct_run` program can be used for running tests from the OS command line, e.g.

- `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
```

It is also possible to combine the `dir`, `suite` and `group/case` flags. E.g, to run `x_SUITE` and `y_SUITE` in directory `testdir`:

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

This has the same effect as calling:

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

For more details on *test case group execution*, please see below.

Other flags that may be used with `ct_run`:

- `-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.
- `-vts`, start web based GUI (see below).
- `-shell`, start interactive shell mode (see below).
- `-step [step_opts]`, step through test cases using the Erlang Debugger (see below).
- `-spec <testspecs>`, use test specification as input (see below).
- `-allow_user_terms`, allows user specific terms in a test specification (see below).
- `-silent_connections [conn_types]`, tells Common Test to suppress printouts for specified connections (see below).
- `-stylesheet <css_file>`, points out a user HTML style sheet (see below).
- `-cover <cover_cfg_file>`, to perform code coverage test (see *Code Coverage Analysis*).
- `-cover_stop <bool>`, to specify if the cover tool shall 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/disable *Built-in Common Test Hooks*. Default is `true`.
- `-include`, specifies include directories (see above).
- `-no_auto_compile`, disables the automatic test suite compilation feature (see above).
- `-abort_if_missing_suites`, aborts the test run if one or more suites fail to compile (see above).
- `-multiply_timetraps <n>`, extends *timetrap timeout* values.
- `-scale_timetraps <bool>`, enables automatic *timetrap timeout* scaling.
- `-repeat <n>`, tells Common Test to repeat the tests n times (see below).
- `-duration <time>`, tells Common Test to repeat the tests for duration of time (see below).
- `-until <stop_time>`, tells Common Test to repeat the tests until stop_time (see below).
- `-force_stop [skip_rest]`, on timeout, the test run will be aborted when current test job is finished. If `skip_rest` is provided the rest of the test cases in the current test job will be skipped (see below).
- `-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 might not be compatible with older browsers.
- `-logopts <opts>`, makes it possible to modify aspects of the logging behaviour, see *Log options* below.
- `-verbosity <levels>`, sets *verbosity levels for printouts*.

> **Note:**
>
> Directories passed to Common Test may have either relative or absolute paths.

> Note:
>
> Arbitrary start flags to the Erlang Runtime System may also be passed as parameters to `ct_run`. It is, for example, useful to be able to pass directories that should be added to the Erlang code server search path with the `-pa` or `-pz` flag. 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 preferrably added to the code path this way. Example:
>
> ```
> $ ct_run -dir ./chat_server -logdir ./chat_server/testlogs -pa $PWD/chat_server/ebin
> ```
>
> Note how in this example, the absolute path of the `chat_server/ebin` directory is passed to the code server. This is essential since relative paths are stored by the code server as relative, and Common Test changes the current working directory of the Erlang Runtime System 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, i.e. one 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 e.g. compilation errors, an illegal return value from an info function, etc.

If auto skipped test cases should not affect the exit status, you may change the default behaviour using start flag:

```
-exit_status ignore_config
```

For more information about the `ct_run` program, see the *Reference Manual* and the *Installation* chapter.

1.7.4 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 called *`ct:run_test/1`*. This function takes the same start parameters as the *`ct_run`* program described above, only the flags are instead given as options in a list of key-value tuples. E.g. a test specified with `ct_run` like:

```
$ 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.

Releasing the Erlang shell

During execution of tests, started with *`ct:run_test/1`*, the Erlang shell process, controlling stdin, will remain the top level process of the Common Test system of processes. The result is that the Erlang shell is not available for interaction during the test run. If this is not desirable, maybe because the shell is needed for debugging purposes or for interaction with the SUT during test execution, you may set the `release_shell` start option to `true` (in the call to `ct:run_test/1` or by using the corresponding test specification term, see below). This will make 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, and 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:**
>
> Note that in order to use the *ct:break/1/2* and *ct:continue/0/1* functions, release_shell *must* be set to true.

For detailed documentation about *ct:run_test/1*, please see the *ct* manual page.

1.7.5 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 is as follows (on the command line):

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

or (in the Erlang shell):

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

The group_names_or_paths parameter specifies either one or more group names and/or one or more group paths. At start up, Common Test will search for matching groups in the group definitions tree (i.e. the list returned from Suite:groups/0, please see the *Test case groups* chapter for details). Given a group name, say g, Common Test will search for all paths that lead to g. By path here we mean a sequence of nested groups, all of which have to be followed in order to get from the top level group to g. Actually, what Common Test needs to do in order to execute the test cases in group g, is to call the init_per_group/2 function for each group in the path to g, as well as all corresponding end_per_group/2 functions afterwards. The obvious reason for this is that 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, etc, all the way up to the top level group.

As you may have already realized, 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 will be performed. Common Test actually interprets a group specification that consists of a single name this way:

"Search and find all paths in the group definitions tree that lead to the specified group and, for each path, create a test which (1) executes all configuration functions in the path to the specified group, then (2) executes all - or all matching - test cases in this group, as well as (3) all - or all matching - test cases in all sub groups of the group".

It is also possible for the user to specify a specific group path with the group_names_or_paths parameter. With this type of specification it's possible to avoid execution of unwanted groups (in otherwise matching paths), and/or the execution of sub groups. The syntax of the group path is a list of group names in the path, e.g. on the command line:

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

or similarly in the Erlang shell (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 will be the terminating group in the test, i.e. no sub groups following this group will be executed. In the example above, g4 is the terminating group, hence Common Test will execute a test that calls all init configuration functions in the path to g4, i.e. g1..g3..g4. It will then call test cases tc1 and tc5 in g4 and finally all end configuration functions in order g4..g3..g1.

Note that the group path specification doesn't necessarily have to include *all* groups in the path to the terminating group. Common Test will search for all matching paths if given an incomplete group path.

Note also that it's possible to combine group names and group paths with the `group_names_or_paths` parameter. Each element is treated as an individual specification in combination with the `cases` parameter. See examples below.

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]}].
```

```
$ ct_run -suite "x_SUITE" -group all
```

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

Two tests will be executed, one for all cases and all sub groups under `top1`, and one for all under `top2`. (We would get the same result with `-group top1 top2`, or `{group,[top1,top2]}`.

```
$ ct_run -suite "x_SUITE" -group top1
```

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

This will execute one test for all cases and sub groups under `top1`.

```
$ ct_run -suite "x_SUITE" -group top1 -case tc12
```

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

This will run a test that executes `tc12` in `top1` and any sub group 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]}]).
```

This will execute `tc12` *only* in group `top1`.

```
$ ct_run -suite "x_SUITE" -group top1 -case tc16
```

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

This will search `top1` and all its sub groups for `tc16` and the result will be that this test case executes in group `sub121`. (The specific path: `-group [sub121]` or `{group,[[sub121]]}`, would have given us the same result in this example).

```
$ ct_run -suite "x_SUITE" -group sub12 [sub12]
```

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

This will execute two tests, one that includes all cases and sub groups under `sub12`, and one with *only* the test cases in `sub12`.

```
$ ct_run -suite "x_SUITE" -group sub2X2
```

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

In this example, Common Test will find and execute two tests, one for the path from `top2` to `sub2X2` via `sub21`, and one from `top2` to `sub2X2` via `sub22`.

```
$ ct_run -suite "x_SUITE" -group [sub21,sub2X2]
```

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

Here, by specifying the unique path: `top2 -> sub21 -> sub2X2`, only one test is executed. The second possible path from `top2` to `sub2X2` (above) will be discarded.

```
$ ct_run -suite "x_SUITE" -group [sub22] -case tc22 tc21
```

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

In this example only the test cases for `sub22` will be executed, and in reverse order compared to the group definition.

If a test case that belongs to a group (according to the group definition), is executed without a group specification, i.e. simply by means of (command line):

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

or (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, exactly as it has been presented in this section, can also be used in *Test Specifications* (with some extra features added). Please see below.

1.7.6 Running the interactive shell mode

You can start Common Test in an interactive shell mode where no automatic testing is performed. Instead, in this mode, 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 e.g. for debugging test suites, for analysing and debugging the SUT during "simulated" test case execution, and for trying out various operations during test suite development.

To invoke the interactive shell mode, you can 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 may start the Erlang shell and Common Test in the same go by using the `-shell` and, optionally, the `-config` and/or `-userconfig` flag. Examples:

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

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

If any functions using "required config data" (e.g. ct_telnet or ct_ftp functions) are to be called from the erlang shell, config data must first be required with *ct:require/1/2*. This is equivalent to a `require` statement in the *Test Suite Info Function* or in the *Test Case Info 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, will in the interactive mode be written to a log named `ctlog.html` in the `ct_run.<timestamp>` directory. A link to this file will be available in the file named `last_interactive.html` in the directory from which you executed `ct_run`. Currently, specifying a different root directory for the logs than the current working directory, is not supported.

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

1.7.7 Step by step execution of test cases with the Erlang Debugger

By means of `ct_run -step [opts]`, or by passing the `{step,Opts}` option to *ct:run_test/1*, it is possible to get the Erlang Debugger started automatically and use its graphical interface to investigate the state of the current test case and to execute it step by step and/or set execution breakpoints.

If no extra options are given with the `step` flag/option, breakpoints will be set automatically on the test cases that are to be executed by Common Test, and those functions only. If the step option `config` is specified, breakpoints will also be initially set on the configuration functions in the suite, i.e. `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 will automatically pop up. (This is because each test case executes on a dedicated Erlang process). Whenever a new test case starts, Common Test will attempt to close the inactive trace window of the previous test case. However, if you prefer that Common Test leaves inactive trace windows, use the `keep_inactive` option.

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

1.7.8 Test Specifications

General description

The most flexible way to specify what to test, is to use a so called test specification. A test specification is a sequence of Erlang terms. The terms are normally declared in one or more text files (see *ct:run_test/1*), but may 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 possible to e.g. label the test run (similar to `ct_run -label`), evaluate arbitrary 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 that should 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`), and more.

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

With test specification terms it is possible to state exactly which tests should 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.

An arbitrary number of test terms may be declared in sequence. Common Test will by default compile the terms into one or more tests to be performed in one resulting test run. Note that a term that specifies a set of test cases will "swallow" one that only specifies a subset of these cases. E.g. the result of merging one term that specifies that all cases in suite S should 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 behaviour, i.e. to instead perform each test sequentially in a "script-like" manner, the term `merge_tests` can be set 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 given at startup (either with `ct_run -spec file1 file2 ...` or `ct:run_test([{spec, [File1,File2,...]}])`), Common Test will either execute one test run per specification file, or join the files and perform all tests within one single test run. The first behaviour is the default one. The latter requires that the start flag/option `join_specs` is provided, e.g. `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 may be combined with) test specification file inclusion, described next.

Test specification file inclusion

With the `specs` term (see syntax below), it's possible to have a test specification include other specifications. An included specification may either be joined with the source specification, or used to produce a separate test run (like with the `join_specs` start flag/option 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" will be joined with the terms in the source specification "a.spec" (if any). The inclusion of specifications "d.spec" and "e.spec" will result in two separate, and independent, test runs (i.e. one for each included specification).

Note that the `join` option does not imply that the test terms will be merged (see `merge_tests` above), 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 configuration that can not 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 the `separate` option, do not share configuration settings with the source specification. This is useful e.g. if there are clashing configuration settings in included specifications, making it impossible to join them.

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

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

The `specs` term may of course be used to nest specifications, i.e. have one specification include other specifications, which in turn include others, etc.

Test case groups

When a test case group is specified, the resulting test executes the `init_per_group` function, followed by all test cases and sub groups (including their configuration functions), and finally the `end_per_group` function. 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 which is defined (in `Suite:group/0`) to be a sub group of another group, is specified (or if particular test cases of a sub group are), Common Test will call the configuration functions for the top level groups as well as for the sub group in question (making it possible to pass configuration data all the way from `init_per_suite` down to the test cases in the sub group).

The test specification utilizes the same mechanism for specifying test case groups by means of names and paths, as explained in the *Group Execution* section above, with the addition of the `GroupSpec` element described next.

The `GroupSpec` element makes it possible to specify group execution properties that will override those in the group definition (i.e. in `groups/0`). Execution properties for sub-groups may be overridden as well. This feature makes it possible to change properties of groups at the time of execution, without even having to edit the test suite. The very same feature is available for `group` elements in the `Suite:all/0` list. Therefore, more detailed documentation, and examples, can be found in the *Test case groups* chapter.

Test specification syntax

Below is the 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 the `init` term are only relevant in the latter (see the *Large Scale Testing* chapter for information). For more information about the various terms, please see the corresponding sections in the User's Guide, such as e.g. the *`ct_run` program* for an overview of available start flags (since most flags have a corresponding configuration term), and more detailed explanation of e.g. *Logging* (for the `verbosity`, `stylesheet` and `basic_html` terms), *External Configuration Data* (for the `config` and `userconfig` terms), *Event Handling* (for the `event_handler` term), *Common Test Hooks* (for the `ct_hooks` term), etc.

Config 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}.
```

```
       {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, i.e. without directory paths. `ConfigFiles` must be full names, including paths. E.g, these 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, will be relative to the directory which contains the test specification file, if `ct_run -spec TestSpecFile ...` or `ct:run:test([{spec,TestSpecFile},...])` executes the test. The path will be relative to the top level log directory, if `ct:run:testspec(TestSpec)` executes the test.

Constants

The `define` term introduces a constant, which is used to replace the name `Constant` with `Value`, wherever it's found in the test specification. This replacement happens during an initial iteration through the test specification. Constants may be used anywhere in the test specification, e.g. in arbitrary 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 upper case letter, or a $, ?, or _. This also means that it must always be single quoted (obviously, since the constant name is actually 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. Compare these terms:

```
%% 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 has been deprecated but will remain supported in upcoming Common Test releases. Replacing `alias` terms with `define` is strongly recommended though! Here's an example of such a replacement:

```
  %% 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}.
```

Actually, constants could well replace the `node` term too, but this still has declarative value, mainly when used in combination with `NodeRefs == all_nodes` (see types above).

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 will be used for storing the HTML log files (in subdirectories tagged with node name, date and time).
- The variables in the specified test system config files will be imported for the test.
- The first test to run includes all suites for system t1. Excluded from the test are however the t1B and t1D suites. Also test cases test3 and test4 in t1A as well as the test1 case in t1C are excluded from the test.
- Secondly, the test for system t2 should run. The included suites are t2B and t2C. Included are also test cases test4, test1 and test7 in suite t2A. Note that the test cases will be executed in the specified order.
- Lastly, all suites for systems t3 are to be completely skipped and this should be explicitly noted in the log files.

The init term

With the `init` term it's possible to specify initialization options for nodes defined in the test specification. Currently, there are options to start the node and/or to evaluate any function on the node. See the *Automatic startup of the test target nodes* chapter for details.

User specific terms

It is possible for the user to provide a test specification that includes (for Common Test) unrecognizable terms. If this is desired, the `-allow_user_terms` flag should be used when starting tests with `ct_run`. This forces Common Test to ignore unrecognizable terms. Note that 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 means of the tuple: `{allow_user_terms,true}`

1.7.9 Running tests from the Web based GUI

The web based GUI, VTS, is started with the *`ct_run`* program. From the GUI you can load config files, and select directories, suites and cases to run. You can also state the config files, directories, suites and cases on the command line when starting the web based GUI.

- `ct_run -vts`
- `ct_run -vts -config <configfilename>`
- `ct_run -vts -config <configfilename> -suite <suitewithfullpath> -case <casename>`

From the GUI you can run tests and view the result and the logs.

Note that `ct_run -vts` will try to open the Common Test start page in an existing web browser window or start the browser if it is not running. Which browser should be started may be specified with the browser start command option:

```
ct_run -vts -browser <browser_start_cmd>
```

Example:

```
$ ct_run -vts -browser 'firefox&'
```

Note that the browser must run as a separate OS process or VTS will hang!

If no specific browser start command is specified, Firefox will be the default browser on Unix platforms and Internet Explorer on Windows. If Common Test fails to start a browser automatically, or `'none'` is specified as the value for -browser (i.e. `-browser none`), start your favourite browser manually and type in the URL that Common Test displays in the shell.

1.7.10 Log files

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

- Text to the operator's 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 gets 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, who may run hundreds or thousands of test cases, doesn't want to fill the console with details about, or printouts from, the specific test cases. By default, the operator will only see:

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

If/when the operator wants to dig deeper into the general results, or the result of a specific test case, he should do so by following the links in the HTML presentation and take a look in the major or minor log files. The "all_runs.html" page is a practical starting point usually. It's located in `logdir` and contains a link to each test run including a quick overview (date and time, node name, number of tests, test names and test result totals).

An "index.html" page is written for each test run (i.e. stored in the "ct_run" directory tagged with node name, date and time). This file gives a short overview of all individual tests performed in the same test run. The test names follow this convention:

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

On the test run index page there is 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 very helpful when analyzing test results or debugging test suites.

On the test run index page it is noted if a test has missing suites (i.e. suites that Common Test has 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 etc. The information is available in both a file with textual and with HTML representation. The HTML file shows a summary which 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 one in a separate file. This way, it should be straightforward to compare the latest results to that of previous test runs, even if the set of test cases changes. If SASL is running, its logs will also be printed to the current minor log file by the *cth_log_redirect built-in hook*.

The full name of the minor log file (i.e. the name of the file including the absolute directory path) can be read during execution of the test case. It comes as value in the 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. Please see the *Event Handling* chapter for details.

Which information goes where is user configurable via the test server controller. Three threshold values determine what comes out on screen, and in the major or minor log files. See the OTP Test Server manual for information. The contents that goes to the HTML log file is fixed however and cannot be altered.

The log files are written continously during a test run and links are always created initially when a test starts. This makes it possible to follow test progress simply by refreshing pages in the HTML browser. Statistics totals are not presented until a test is complete however.

Log options

With the `logopts` start flag, it's possible to specify options that modify some aspects of the logging behaviour. Currently, the following options are available:

- `no_src`
- `no_nl`

With `no_src`, the html version of the test suite source code will not be generated during the test run (and consequently not be available in the log file system).

With `no_nl`, Common Test will not add a newline character (\n) to the end of an output string that it receives from a call to e.g. `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_src
```

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

```
xxx
```

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

Sorting HTML table columns

By clicking the name in the column header of any table (e.g. "Ok", "Case", "Time", etc), the table rows are sorted in whatever order makes sense for the type of value (e.g. numerical for "Ok" or "Time", and alphabetical for "Case"). The sorting is performed by means of 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

On the test suites overview page you find a link to the Unexpected I/O Log. In this log, Common Test saves printouts made with `ct:log/2` and `ct:pal/2`, as well as captured system error- and progress reports, that cannot be associated with particular test cases and therefore cannot be written to individual test case log files. This happens e.g. 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 parallell test case group.

The Pre- and Post Test I/O Log

On the Common Test Framework Log page you find links to the so called Pre- and Post Test I/O Log. In this log, Common Test saves printouts made with `ct:log/2` and `ct:pal/2`, as well as captured system error- and progress reports, that take place before - and after - the actual 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 due to 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, please see the *Synchronizing* section in the Common Test Hooks chapter.

Note that logging to file with `ct:log/2` or `ct:pal/2` only works when Common Test is running. Printouts with `ct:pal/2` are however always displayed on screen.

1.7.11 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, for some reason, 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, as well as JavaScripts (see table sorting above).

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` and `pal/3`) accept `Category` as first argument. With this argument it's possible to specify a category that can be mapped to a selector in a CSS definition. This is useful especially for coloring text differently depending on the type of (or reason for) the printout. Say you want one 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 may look like this:

```
div.sys_config  { background:blue; color:white }
div.sys_state   { background:yellow; color:black }
div.error       { background:red; color:white }
```

To install the CSS file (Common Test inlines the definition in the HTML code), the name may 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 the `-stylesheet` flag are on a global test level in the sense that they can be used in any suite which is part of the test run.

It is also possible to install style sheets 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 can not be used by other suites. A suite private style sheet, if specified, will be used in favour of a global style sheet (one specified with the `-stylesheet` flag). A stylesheet tuple (as returned by `suite/0` above) can also be returned from a test case info 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 favour of a suite or global level style sheet.

In a tuple `{stylesheet,CSSFile}`, if `CSSFile` is specified with a path, e.g. `"$TEST/styles/categories.css"`, this full name will be used to locate the file. If only the file name is specified however, e.g.

"categories.css", then the CSS file is assumed to be located in the data directory, `data_dir`, of the suite. The latter usage is recommended since it is portable compared to hard coding path names in the suite!

The `Category` argument in the example above may have the value (atom) `sys_config` (white on blue), `sys_state` (black on yellow) or `error` (white on red).

1.7.12 Repeating tests

You can order Common Test to repeat the tests you specify. You can choose to repeat tests a certain 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 means of time, it is also possible to specify what action Common Test should take upon timeout. Either Common Test performs all tests in the current run before stopping, or it stops as soon as the current test job is finished. Repetition can be activated by means of `ct_run` start flags, or tuples in the `ct:run:test/1` option list argument. The flags (options in parenthesis) are:

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

The duration time, `DurTime`, is specified as `HHMMSS`. Example: `-duration 012030` or `{duration,"012030"}`, means the tests will be executed and (if time allows) repeated, until timeout occurs after 1 h, 20 min and 30 secs. `StopTime` can be specified as `HHMMSS` and is then interpreted as a time today (or possibly tomorrow). `StopTime` can also be specified as `YYMoMoDDHHMMSS`. Example: `-until 071001120000` or `{until,"071001120000"}`, which means the tests will be executed and (if time allows) repeated, until 12 o'clock on the 1st of Oct 2007.

When timeout occurs, Common Test will never abort the ongoing test case, since this might leave the system under test in an undefined, and possibly bad, state. Instead Common Test will by default finish the current test run before stopping. If the `force_stop` flag is given, Common Test will stop as soon as the current test job is finished, and if the `force_stop` flag is given with `skip_rest` Common Test will only complete the current test case and skip the rest of the tests in the test job. Note that since Common Test always finishes off at least the current test case, the time specified with `duration` or `until` is never definitive!

Log files from every single repeated test run is saved in normal Common Test fashion (see above). Common Test may 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.

Note that 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. There you can read the repetition number, remaining time, etc.

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, will be executed in one test run. A timeout event will occur after 10 minutes. As long as there is time left, Common Test will repeat the test run (i.e. starting over with the to1 test). When the timeout occurs, Common Test will stop as soon as the current job is finished (because of the `force_stop` flag). As a result, the specified test run might be aborted after the to1 test and before the to2 test.

Example 2:

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

Here the same test run as in Example 1, but with the `force_stop` flag set to `skip_rest`. If the timeout occurs while executing tests in directory to1, the rest of the test cases in to1 will be skipped and then the test will be aborted without running the tests in to2 another time. If the timeout occurs while executing tests in directory to2, then the rest of the test cases in to2 will be skipped and then the test will be 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 example above will be executed (and possibly repeated). In this example, however, the timeout will occur after 1 hour and when that happens, Common Test will finish the entire test run before stopping (i.e. the to1 and to2 test will always both be 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, will be repeated 5 times.

> ### Note:
>
> This feature should not be confused 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 the *Writing Test Suites* chapter.

1.7.13 Silent Connections

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

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

where `conn_types` specifies `ssh, telnet, ftp, rpc` and/or `snmp`.

Example:

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

switches off logging for ssh and telnet connections.

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

switches off logging for all connection types.

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

It is possible to also specify `silent_connections` in a test suite. This is accomplished by returning a tuple, `{silent_connections,ConnTypes}`, in the `suite/0` or test case info list. If `ConnTypes` is a list of atoms (`ssh, telnet, ftp, rpc` and/or `snmp`), output for any corresponding connections will be suppressed. Full logging is per 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:

```
-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 should be silent. The result is that `my_testcase1` will get telnet info (if any) printed in the log, but not ssh info. `my_testcase2` will get no info from either connection printed.

`silent_connections` may also be specified with a term in a test specification (see *Test Specifications*). Connections provided with the `silent_connections` start flag/option, will be merged with any connections listed in the test specification.

The `silent_connections` start flag/option and test specification term, overrides any settings made by the info functions inside the test suite.

Note:

Note that in the current Common Test version, the `silent_connections` feature only works for telnet and ssh connections! Support for other connection types will be added in future Common Test versions.

1.8 External Configuration Data

1.8.1 General

To avoid hard coding data values related to the test and/or SUT (System Under Test) in the test suites, the data may instead be specified by means of 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 having to modify the actual test suites using the data. Examples of configuration data:

- Addresses to the test plant or other instruments
- User login information
- Names of files needed by the test
- Names of programs that should be executed during the test
- Any other variable needed by the test

1.8.2 Syntax

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

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

where

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

1.8.3 Requiring and reading configuration data

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

`require` is an assert statement that can be part of the *test suite info function* or *test case info function*. If the required variable is not available, the test is skipped (unless a default value has been specified, see the *test case info function* chapter for details). There is also a function `ct:require/1/2` which can be called from a test case in order 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 (e.g. to skip the test case if the variable in question doesn't exist).

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

- They may be introduced to identify connections (see below).
- They may used to help adapt configuration data to a test suite (or test case) and improve readability.

To read the value of a config variable, use the function `get_config/1/2/3` which is also described in the reference manual for *ct*.

Example:

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

    ...

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

1.8.4 Using configuration variables defined in multiple files

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

1.8.5 Encrypted configuration files

It is possible to encrypt configuration files containing sensitive data if these files must be stored in open and shared directories.

Call *`ct:encrypt_config_file/2/3`* to have Common Test encrypt a specified file using the DES3 function in the OTP `crypto` application. The encrypted file can then be used as a regular configuration file, in combination with other encrypted files or normal text files. The key for decrypting the configuration file must be provided when running the test, however. This can be done by means of the `decrypt_key` or `decrypt_file` flag/option, 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.

Please see the *ct* reference manual for more information.

1.8.6 Opening connections by using configuration data

There are two different methods for opening a connection by means of the support functions in e.g. *`ct_ssh`*, *`ct_ftp`*, and *`ct_telnet`*:

- 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 may be used as connection identity in all subsequent calls related to the connection (also for closing it). It's only possible to have one open connection per target name. If attempting to open a new connection using a name already associated with an open connection, Common Test will return the already existing handle so that the previously opened connection will be used. This is a practical feature since it makes it possible to call the function for opening a particular connection whenever useful. An action like this will not necessarily open any new connections unless it's required (which could be the case if e.g. the previous connection has been closed unexpectedly by the server). Another benefit of using named connections is that it's not necessary 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 will result in multiple connections being opened. This can be useful e.g. 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).

1.8.7 User specific configuration data formats

It is possible for the user to specify configuration data on a different format than key-value tuples in a text file, as described so far. The data can e.g. be read from arbitrary 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

The Common Test application includes default callback modules for handling configuration data specified in standard config files (see above) and in xml files:

- `ct_config_plain` - for reading configuration files with key-value tuples (standard format). This handler will be 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

This is an example of an XML configuration file:

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

This configuration file, once read, will produce the same configuration variables as the following text file:

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

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

How to implement a user specific handler

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

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

To perform validation of the configuration strings, the callback module should have the following function exported:

`Callback:check_parameter/1`

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

The return value should be any of the following values indicating if given configuration parameter is valid:

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

To perform reading of configuration data - initially before the tests start, or as a result of data being reloaded during test execution - the following function should be exported from the callback module:

`Callback:read_config/1`

The input argument is the same as for the `check_parameter/1` function.

The return value should be either:

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

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

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

1.8.8 Examples of configuration data handling

A config file for using the FTP client to access files on a remote host could look like this:

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

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

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

Example of how to assert that the configuration data is available and use it 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),
    ...
```

An example of how the above functions could be rewritten if 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),
        ...
```

1.8.9 Example of user specific configuration handler

A simple configuration handling driver which will ask an external server for configuration data can be implemented this way:

```
-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 may be "config_server", if the config_server.erl module below 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.
```

In this example, the handler also provides the ability to dynamically reload configuration variables. If `ct:reload_config(localtime)` is called from the test case function, all variables loaded with `config_driver:read_config/1` will be updated with their latest values, and the new value for variable `localtime` will be returned.

1.9 Code Coverage Analysis

1.9.1 General

Although Common Test was created primarly for the purpose of 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 by means of 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, etc). All the Common Test user needs to do is to specify the extent of the code coverage analysis.

1.9.2 Usage

To specify what modules should be included in the code coverage test, you provide a cover specification file. Using this file you can point out specific modules or specify directories that contain modules which should all be included in the analysis. You can also, in the same fashion, specify modules that should 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 an Erlang node other than the one Common Test is running on. If this is the case you need to specify these other nodes in the cover specification file or add them dynamically to the code coverage set of nodes. See the `ct_cover` page in the reference manual for details on the latter.

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 you per module and total coverage percentages, as well as one HTML file printed for each module included in the analysis that shows exactly what parts of the code have been executed during the test. In overview mode, only the code coverage overview page gets 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 will export collected coverage data to this file at the end of the test. You may similarly specify that previously exported data should be imported and included in the analysis for a test (you can specify multiple import files). This way it is possible to analyse total code coverage without necessarily running all tests at once.

To activate the code coverage support, you simply specify the name of the cover specification file as you start Common Test. This you do either by using the `-cover` flag with `ct_run`. Example:

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

You may also pass the cover specification file name in a call to *`ct:run_test/1`*, by adding a `{cover,CoverSpec}` tuple to the `Opts` argument. Also, you can of course enable code coverage in your test specifications (read more in the chapter about *using test specifications*).

1.9.3 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 in to the test node. If a process at this point is still running 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 will now be killed. To avoid this it is possible to set the value of the `cover_stop` option to `false`. This means that the modules will stay cover compiled, and it is therefore only recommended if the erlang node(s) under test is terminated after the test is completed or if cover can be manually stopped.

The option can be set by using the `-cover_stop` flag with `ct_run`, by adding `{cover_stop,true|false}` to the Opts argument to *`ct:run_test/1`*, or by adding a `cover_stop` term in your test specification (see chapter about *test specifications*).

1.9.4 The cover specification file

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

Note: Directories containing Erlang modules that are to be included in a code coverage test must exist in the code server path, or the cover tool will fail to recompile the modules. (It is not sufficient to specify these directories in the cover specification file for Common Test).

1.9.5 Cross cover analysis

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

This can of course also be achieved in a more customized way by using the `export` parameter in the cover specification and analysing the result off line, but the cross cover mechanism is a build in solution which also provides the logging.

The mechanism is easiest explained via an example:

Let's say that there are two systems, `s1` and `s2`, which are tested in separate test runs. System `s1` contains a library module `m1` which is tested by the `s1` test run and is included in `s1`'s cover specification:

```
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, let's imagine that since `m1` is a library module, it is also used quite a bit by system `s2`. The `s2` test run does not specifically test `m1`, but it might still be interesting to see which parts of `m1` is actually covered by the `s2` tests. To do this, `m1` could be included also in `s2`'s cover specification:

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

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

If instead the cover specification for `s2` was like this:

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

then `m1` would be cover compiled in the `s2` test run, 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` would be available in the cross cover log for the `s1` test run.

The call to the analyse function must be like this:

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

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

Note 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 point of these are only to map the modules specified in the cover specification to the log directory specified in the call to the analyse function. The name of the tag has no meaning beyond this.

1.9.6 Logging

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

Prior to 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 via 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. See the release notes for more information about this change.

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

If cross cover analysis has been performed, and there are accumulated coverage results for the current test, then the - "Coverdata collected over all tests" link will take you to these results.

1.10 Using Common Test for Large Scale Testing

1.10.1 General

Large scale automated testing requires running multiple independent test sessions in parallel. This is accomplished by running a number of 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, CT offers a master test node component, CT Master, that handles central configuration and control in a system of distributed CT nodes.

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

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

The CT master API is exported by the `ct_master` module.

1.10.2 Usage

CT Master requires all test nodes to be on the same network and share a common file system. As of this date, CT Master can not start test nodes automatically. The nodes must have been started in advance for CT 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. In case of a list, the specifications will be 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 will be merged into one combined specification prior to test execution. For example:

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

means first the tests specified by "ts1" will run, 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. The `run/3` function runs the tests in `TestSpecs` just like `run/1` but will also take any test in `TestSpecs` that's not explicitly tagged with a particular node name and execute it on the nodes listed in `InclNodes`. By using `run/3` this way it is possible to use any test specification, with or without node information, in a large scale test environment! `ExclNodes` is a list of nodes that should be excluded from the test. I.e. tests that have been specified in the test specification to run on a particular node will not be performed if that node is at runtime listed in `ExclNodes`.

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

When tests start, CT Master prints information to console about the nodes that are involved. CT Master also reports when tests finish, successfully or unsuccessfully. If connection is lost to a node, the test on that node is considered finished. CT Master will not attempt to reestablish contact with the failing node. At any time to get the current status of the test nodes, call the function:

`ct_master:progress()`

To stop one or more tests, use:

`ct_master:abort()` (stop all) or `ct_master:abort(Nodes)`

For detailed information about the `ct_master` API, please see the *manual page* for this module.

1.10.3 Test Specifications

The test specifications used as input to CT Master are fully compatible with the specifications used as input to the regular CT server. The syntax is described in the *Running Test Suites* chapter.

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:

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

where

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

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

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

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

Consider the example in the *Running Test Suites* chapter, now extended with node information and intended to be executed by the CT 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)`, the t1 test will be executed on node `ct_node@host_x` (node1), the t2 test on `ct_node@host_y` (node2) and the t3 test on both node1 and node2. The t1 config file will only be read on node1 and the t2 config file only on node2, while the t3 config file will be read on both node1 and node2. Both test nodes will write log files to the same directory. (The CT Master node will however use 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 the t1 test does not run on `ct_node@host_x` (or any other node) while the t3 test runs on `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 the *Running Test Suites* chapter). 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), will be performed. Tests without explicit node association will always be performed too of course!

1.10.4 Automatic startup of test target nodes

Is is possible to automatically start, and perform initial actions, on test target nodes by using the test specification term `init`.

Currently, two sub-terms 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` will be started with the default callback module `ct_slave`. The given user name and password is used to log into remote host `host2`. Also, the function `module:function/0` will be evaluated on `node1@host3`, and the result of this call will be printed to the log.

The default *ct_slave* callback module, which is part of the Common Test application, has the following features:

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

Note that it is possible to specify an `eval` term for the node as well as `startup_functions` in the `node_start` options list. In this case first the node will be started, then the `startup_functions` are executed, and finally functions specified with `eval` are called.

1.11 Event Handling

1.11.1 General

It is possible for the operator of a Common Test system to receive event notifications continously during a test run. It is reported e.g. when a test case starts and stops, what the current count of successful, failed and skipped cases is, etc. This information can be used for different purposes such as logging progress and results on other format than HTML, saving statistics to a database for report generation and test system supervision.

Common Test has a framework for event handling which is based on the OTP event manager concept and gen_event behaviour. When the Common Test server starts, it spawns an event manager. During test execution the manager gets a notification from the server every time something of potential interest happens. Any event handler plugged into the event manager can match on events of interest, take action, or maybe simply pass the information on. Event handlers are Erlang modules implemented by the Common Test user according to the gen_event behaviour (see the OTP User's Guide and Reference Manual for more information).

As already described, a Common Test server always starts an event manager. The server also plugs in a default event handler which has as its only purpose to relay notifications to a globally registered CT Master event manager (if a CT Master server is running in the system). The CT Master also spawns an event manager at startup. Event handlers plugged into this manager will receive the events from all the test nodes as well as information from the CT Master server itself.

User specific event handlers may be plugged into a Common Test event manager, either by telling Common Test to install them before the test run (see below), or by adding the handlers dynamically during the test run by means of `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 CT Master node) `ct_master:get_event_mgr_ref/0`.

1.11.2 Usage

Event handlers may be installed by means of 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
```

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

All event handler modules must have gen_event behaviour. Note also that these modules must be precompiled, and that their locations must be added explicitly to the Erlang code server search path (like in the example).

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

```
{event_handler,EventHandlers}

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

Example:

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

This will install two event handlers for the `my_SUITE` test. 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 by means of *test specification* terms:

`{event_handler, EventHandlers}`, or

`{event_handler, EventHandlers, InitArgs}`, or

`{event_handler, NodeRefs, EventHandlers}`, or

`{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 given).

To plug a handler into the CT Master event manager, specify `master` as the node in `NodeRefs`.

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

`#event{name, node, data}`

`name` is the label (type) of the event. `node` is the name of the node the event has originated from (only relevant for CT Master event handlers). `data` is specific for the particular event.

General events:

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

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

  Indicates that the logging process of Common Test has started successfully and is ready to receive IO messages.
- `#event{name = stop_logging, data = []}`

  Indicates that the logging process of Common Test has been 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 will begin executing test cases.
- `#event{name = test_done, data = EndTime}`

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

  This 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()`, the number of tests.

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

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

  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". Note that if a test case group with a repeat property exists in any test, the total number of test cases can not 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 is starting, the group name and execution properties are also given.

- `#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 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 (i.e. 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`, the reason why the case has been 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 has failed.

  `FailInfo = {timetrap_timeout,integer()} | {RunTimeError,StackTrace} | UserTerm`, detailed information about an error.

  `RunTimeError = term()`, a run-time error, e.g. badmatch, undef, etc.

  `StackTrace = list()`, list of function calls preceeding a run-time error.

  `UserTerm = term()`, arbitrary data specified by user, or `exit/1` info.

  This event informs about the end of a test case or a configuration function (see the `tc_start` event for details on the FuncOrGroup element). 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 of course possible to dig deeper and also perform pattern matching on the various reasons for skipped or failed. Note that `{'EXIT',Reason}` tuples have been translated into `{error,Reason}`. Note also that if a `{failed,`

`{Suite,end_per_testcase,FailInfo}` result is received, it actually means the test case was successful, but that `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 has failed.

  `ConfigFunc = init_per_suite | init_per_group`

  `FailInfo = {timetrap_timeout,integer()} | {RunTimeError,StackTrace} | bad_return | UserTerm`, detailed information about an error.

  `FailedCaseInSequence = atom()`, name of a case that has failed in a sequence.

  `RunTimeError = term()`, a run-time error, e.g. badmatch, undef, etc.

  `StackTrace = list()`, list of function calls preceeding a run-time error.

  `UserTerm = term()`, arbitrary data specified by user, or `exit/1` info.

  This event gets 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. Note that this event is never received as a result of a test case getting skipped because of `init_per_testcase` failing, since that information is carried with the `tc_done` event. If a failed test case belongs to a test case group, the second data element is a tuple `{FuncName,GroupName}`, otherwise simply 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()`, reason for skipping the test case.

  This event specifies that a test case has been skipped by the user. It is only ever received if the skip was declared in a test specification. Otherwise, user skip information is received as a `{skipped,SkipReason}` result in the `tc_done` event 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 simply the function name.

- `#event{name = test_stats, data = {Ok,Failed,Skipped}}`

  `Ok = integer()`, the current number of successful test cases.

  `Failed = integer()`, the current number of failed test cases.

  `Skipped = {UserSkipped,AutoSkipped}`

`UserSkipped = integer()`, the current number of user skipped test cases.

`AutoSkipped = integer()`, the current number of auto skipped test cases.

This is a statistics event with the current count of successful, skipped and failed test cases so far. This event gets sent after the end of each test case, immediately following the `tc_done` event.

Internal events:

- `#event{name = start_make, data = Dir}`

  `Dir = string()`, running make in this directory.

  An internal event saying that Common Test will start compiling modules in directory `Dir`.
- `#event{name = finished_make, data = Dir}`

  `Dir = string()`, finished running make in this directory.

  An internal event saying that Common Test is finished compiling modules in directory `Dir`.
- `#event{name = start_write_file, data = FullNameFile}`

  `FullNameFile = string(), full name of the file.`

  An internal event 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.`

  An internal event used by the Common Test Master process to synchronize particular file operations.

The events are also documented in `ct_event.erl`. This module may serve as an example of what an event handler for the CT event manager can look like.

> Note:
>
> To ensure that printouts to standard out (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 syncronize with the Common Test server by matching on the `tc_start` and `tc_done` events. In the period between these events, all IO gets directed to the test case log file. These events are sent synchronously to avoid potential timing problems (e.g. that the test case log file gets closed just before an IO message from an external process gets through). Knowing this, you need to be careful that your `handle_event/2` callback function doesn't stall the test execution, possibly causing unexpected behaviour as a result.

1.12 Dependencies between Test Cases and Suites

1.12.1 General

When creating test suites, it is strongly recommended to not create dependencies between test cases, i.e. letting test cases depend on the result of previous test cases. There are various reasons for this, for example:

- It makes it impossible to run test cases individually.
- It makes it impossible to run test cases in different order.
- It makes debugging very difficult (since a fault could be the result of a problem in a different test case than the one failing).
- There exists no good and explicit ways to declare dependencies, so it may be very 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 may alter the state of the system and for some other test case to run properly, the 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 (i.e. let the test case run if the state is as expected, otherwise reset or fail) and/or 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 may of course also be separately tested if necessary, to ensure they are working as expected). It is sometimes also possible, but not always desirable, to group tests together in one test case, i.e. let a test case perform a "scenario" test (a test that consists 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. We can't configure the server if it hasn't first been started, we can't connect a client until the server has been properly configured, etc. If we want to have one test case for each of the functions, we might be tempted to try to always run the test cases in the stated order and carry possible data (identities, handles, etc) between the cases and therefore introduce dependencies between them. To avoid this we could consider starting and stopping the server for every test. We would implement the start and stop action as common functions that may be called from init_per_testcase and end_per_testcase. (We would of course test the start and stop functionality separately). The configuration could perhaps also be implemented as a common function, maybe grouped with the start function. Finally the testing of connecting and disconnecting a client may be grouped into one test case. The resulting suite would look something like this:

```
-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.
```

1.12.2 Saving configuration data

There might be situations where it is impossible, or infeasible at least, to implement independent test cases. Maybe it is simply not possible to read the SUT state. Maybe resetting the SUT is impossible and it takes much too long 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 may 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) may save the current value of Config - or any list of key-value tuples - so that it can be read by the next executing test case (or test suite). 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 the 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 the `config` macro with a `saved_config` key:

`{Saver,ConfigList} = ?config(saved_config, Config)`

`Saver` (`atom()`) is the name of the previous test case (where the data was saved). The `config` macro may 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 due to restructuring of the test suite may 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 should be saved by the `end_per_suite` function and read by `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).
```

It is also possible to save `Config` data from a test case that is to be skipped. To accomplish this, return the following tuple:

`{skip_and_save,Reason,ConfigList}`

The result will be that the test case is skipped with `Reason` printed to the log file (as described in previous chapters), and `ConfigList` is saved for the next test case. `ConfigList` may be read by means of `?config(saved_config, Config)`, as described above. `skip_and_save` may also be returned from `init_per_suite`, in which case the saved data can be read by `init_per_suite` in the suite that follows.

1.12.3 Sequences

It is possible that test cases depend on each other so that if one case fails, the following test(s) should not be executed. Typically, if the `save_config` facility is used and a test case that is expected to save data crashes, the following case can not run. CT 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 by means of the `groups/0` function in the test suite (see the *Test case groups* chapter for details).

For example, if we would like to make sure that if `allocate` in `server_b_SUITE` (above) crashes, `deallocate` is skipped, we may define a sequence like this:

```
groups() -> [{alloc_and_dealloc, [sequence], [alloc,dealloc]}].
```

Let's also assume the suite contains the test case `get_resource_status`, which is independent of the other two cases, then the `all` function could look like this:

```
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` will run no matter what happens to the `alloc_and_dealloc` cases.

Test cases in a sequence will be executed in order until they have all succeeded or until one case 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. An arbitrary number of sequences may be specified. Example:

```
groups() -> [{scenarioA, [sequence], [testA1, testA2]},
             {scenarioB, [sequence], [testB1, testB2, testB3]}].

all() -> [test1,
          test2,
          {group,scenarioA},
          test3,
          {group,scenarioB},
          test4].
```

It is possible to have sub-groups in a sequence group. Such sub-groups can have any property, i.e. they are not required to also be sequences. If you want the status of the sub-group to affect the sequence on the level above, return `{return_group_result,Status}` from `end_per_group/2`, as described in the *Repeated groups* chapter. A failed sub-group (`Status == failed`) will cause the execution of a sequence to fail in the same way a test case does.

1.13 Common Test Hooks

1.13.1 General

The *Common Test Hook* (henceforth called CTH) framework allows extensions of the default behaviour of Common Test by means of hooks before and after all test suite calls. CTHs allow advanced Common Test users to abstract out behaviour which is common to multiple test suites without littering all test suites with library calls. Some example usages are: logging, starting and monitoring external systems, building C files needed by the tests and much more!

In brief, Common Test Hooks allows you to:

- Manipulate the runtime config before each suite configuration call
- Manipulate the return of all suite configuration calls and in extension the result of the test themselves.

The following sections describe how to use CTHs, when they are run and how to manipulate your test results in a CTH

> Warning:
>
> When executing within a CTH all timetraps are shutoff. So if your CTH never returns, the entire test run will be stalled!

1.13.2 Installing a CTH

There are multiple ways to install a CTH 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 different ways to accomplish that.

- 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`, i.e. `ct_run -ct_hooks cth1 [{debug,true}] and cth2 ...`.
- Add the `ct_hooks` tag to your *Test Specification*
- Add the `ct_hooks` tag to your call to *ct:run_test/1*

You can also add CTHs within a test suite. This is done by returning `{ct_hooks,[CTH]}` in the config list from *suite/0*, *init_per_suite/1* or *init_per_group/2*. `CTH` in this case can be either 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. Eg: `{ct_hooks,[my_cth_module]}` or `{ct_hooks,[{my_cth_module,[{debug,true}]}]}` or `{ct_hooks,[{my_cth_module,[{debug,true}],500}]}`

Overriding CTHs

By default each installation of a CTH will cause a new instance of it to be activated. This can cause problems if you want to be able to override CTHs in test specifications while still having them in the suite info function. The *id/1* callback exists to address this problem. By returning the same `id` in both places, Common Test knows that this CTH has already been installed and will not try to install it again.

CTH Execution order

By default each CTH installed will be executed in the order which they are installed for init calls, and then reversed for end calls. This is not always wanted so common_test allows the user to specify a priority for each hook. The priority can either be specified in the CTH *init/2* function or when installing the hook. The priority given at installation will override the priority returned by the CTH.

1.13.3 CTH Scope

Once the CTH is installed into a certain test run it will be there until its scope is expired. The scope of a CTH depends on when it is installed. The *init/2* is called at the beginning of the scope and the *terminate/1* function 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 to be run.	the last test suite has been run.
Test Specification	the first test suite is to be 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/4* is called.	*post_end_per_group/4* has been called for that group.

Table 13.1: Scope of a CTH

CTH Processes and Tables

CTHs are run with the same process scoping as normal test suites i.e. a different process will execute 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 will exit after the post hook ends. Also if you for some reason need an ETS table with your CTH, you will have to spawn a process which handles it.

External configuration data and Logging

It's possible in the CTH to read configuration data values by calling `ct:get_config/1/2/3` (as explained in the *External configuration data* chapter). The config variables in question must, as always, first have been `required` by means of a suite-, group-, or test case info function, or the `ct:require/1/2` function. Note that the latter can also be used in CT hook functions.

The CT hook functions may call any of the logging functions available in the `ct` interface to print information to the log files, or to add comments in the suite overview page.

1.13.4 Manipulating tests

It is through CTHs possible to manipulate the results of tests and configuration functions. The main purpose of doing this with CTHs is to allow common patterns to be abstracted out from test test suites and applied to multiple test suites without duplicating any code. All of the callback functions for a CTH follow a common interface, this interface is described below.

Common Test will always call 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, ...)` will be called for test suite `x_SUITE`, even if it doesn't export `init_per_suite/1`. This feature makes it possible to use hooks as configuration fallbacks, or even completely replace all configuration functions with hook functions.

Pre Hooks

It is possible in a CTH to hook in behaviour before *init_per_suite*, *init_per_group*, *init_per_testcase*, *end_per_group* and *end_per_suite*. This is done in the CTH functions called pre_<name of function>. All of these functions take the same three arguments: `Name`, `Config` and `CTHState`. The return value of the CTH function is always a combination of an result for the suite/group/test and an updated `CTHState`. If you want the test suite to continue on executing you should return the config list which you want the test to use as the result. If you for some reason want to skip/fail the test, return 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 using multiple CTHs, the first part of the return tuple will be used as input for the next CTH. So in the case above the next CTH might get `{fail,Reason}` as the second parameter. If you have many CTHs which interact, it might be a good idea to not let each CTH return `fail` or `skip`. Instead return that an action should be taken through the `Config` list and implement a CTH which at the end takes the correct action.

Post Hooks

It is also possible in a CTH to hook in behaviour after *init_per_suite*, *init_per_group*, *end_per_testcase*, *end_per_group* and *end_per_suite*. This is done in the CTH functions called post_<name of function>. All of these function take the same four arguments: `Name`, `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 failed by crashing, `Return` will be `{'EXIT',{{Error,Reason},Stacktrace}}`.

The return value of the CTH function is always a combination of an result for the suite/group/test and an updated `CTHState`. If you want the callback to not affect the outcome of the test you should return the `Return` data as it is given to the CTH. You can also modify the result of the test. By returning the `Config` list with the `tc_status` element 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(_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(_TC, Config, Return, CTHState) ->
  %% Do nothing if tc does not crash.
  {Return, CTHState}.
```

> Note:
>
> Recovering from a testcase failure using CTHs should only be done as a last resort. If used wrongly it could become very difficult to determine which tests 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* might be called if the testcase failed or was skipped respectively. You cannot affect the outcome of the tests any further at this point.

1.13.5 Synchronizing external user applications with Common Test

CTHs can be used to synchronize test runs with external user applications. The init function may e.g. start and/or communicate with an application that has the purpose of preparing the SUT for an upcoming test run, or maybe initialize a database for saving test data to during the test run. The terminate function may 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 will be 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`).

In order to ensure that Common Test doesn't start executing tests, or closes its log files and shuts down, before the external application is ready for it, Common Test may 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 in order to return from the `receive`. These macros are defined in `ct.hrl`.

1.13.6 Example CTH

The CTH below will log information about a test run into a format parseable by *file:consult/1*.

```
%%% @doc 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/3]).
-export([post_init_per_group/4]).
-export([pre_end_per_group/3]).
-export([post_end_per_group/4]).

-export([pre_init_per_testcase/3]).
-export([post_end_per_testcase/4]).

-export([on_tc_fail/3]).
-export([on_tc_skip/3]).

-export([terminate/1]).

-record(state, { file_handle, total, suite_total, ts, tcs, data }).

%% @doc Return a unique id for this CTH.
id(Opts) ->
  proplists:get_value(filename, Opts, "/tmp/file.log").

%% @doc 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 = [] }}.

%% @doc Called before init_per_suite is called.
pre_init_per_suite(Suite,Config,State) ->
    {Config, State#state{ suite_total = 0, tcs = [] }}.

%% @doc Called after init_per_suite.
post_init_per_suite(Suite,Config,Return,State) ->
    {Return, State}.

%% @doc Called before end_per_suite.
pre_end_per_suite(Suite,Config,State) ->
    {Config, State}.

%% @doc 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 } }.

%% @doc Called before each init_per_group.
pre_init_per_group(Group,Config,State) ->
    {Config, State}.

%% @doc Called after each init_per_group.
post_init_per_group(Group,Config,Return,State) ->
    {Return, State}.

%% @doc Called after each end_per_group.
pre_end_per_group(Group,Config,State) ->
    {Config, State}.

%% @doc Called after each end_per_group.
post_end_per_group(Group,Config,Return,State) ->
    {Return, State}.

%% @doc Called before each test case.
pre_init_per_testcase(TC,Config,State) ->
    {Config, State#state{ ts = now(), total = State#state.suite_total + 1 } }.

%% @doc Called after each test case.
post_end_per_testcase(TC,Config,Return,State) ->
    TCInfo = {testcase, TC, Return, timer:now_diff(now(), State#state.ts)},
    {Return, State#state{ ts = undefined, tcs = [TCInfo | State#state.tcs] } }.

%% @doc 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(TC, Reason, State) ->
    State.

%% @doc Called when a test case is skipped by either user action
%% or due to an init function failing.
on_tc_skip(TC, Reason, State) ->
    State.

%% @doc 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.
```

1.13.7 Built-in CTHs

Common Test is delivered with a couple of general purpose CTHs that can be enabled by the user to provide some generic testing functionality. Some of these are enabled by default when starting running common_test, they can be disabled by setting `enable_builtin_hooks` to `false` on the command line or in the test specification. In the table below there is a list of all current CTHs which are delivered with Common Test.

CTH Name	*Is Built-in*	*Description*
cth_log_redirect	yes	Captures all error_logger and SASL logging events and prints them to the current test case log. If an event can not be associated with a testcase it will be printed in the common test

		framework log. This will happen for testcases which are run in parallel and events which occur inbetween testcases. You can configure the level of *SASL* events report using the normal SASL mechanisms.

cth_surefire	no	Captures all test results and outputs them as surefire XML into a file. The file which is created is by default called junit_report.xml. The file name can be changed by setting the `path` option for this hook, e.g.
		`-ct_hooks cth_surefire [{path,"/tmp/report.xml`
		If the `url_base` option is set, an additional attribute named `url` will be added to each `testsuite` and `testcase` XML element. The value will be constructed from the `url_base` and a relative path to the test suite or test case log respectively, e.g.
		`-ct_hooks cth_surefire [{url_base, "http://mys`
		will give a url attribute value similar to
		`"http://myserver.com/ct_run.ct@myhost.2012-12-`
		`x86_64-unknown-linux-gnu.my_test.logs/run.2012`
		Surefire XML can for instance be used by Jenkins to display test results.

1.14 Some thoughts about testing

1.14.1 Goals

It's 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 may be viable options for those that wish 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 didn't know before.

1.14.2 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 other do the same, but uses all kinds of illegal parameters, and verifies 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, some test suites even test internal functions with help of special load-modules on target.

Another interesting category of test suites are the ones that check that fixed bugs don't reoccur. When a bugfix is introduced, a test case that checks for that specific bug should be written and submitted to the affected test suite(s).

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 a lot more expensive than others.

Aim for functionality testing rather than implementation details. Implementation details change quite often, and the test suites should be long lived. Often implementation details differ on different platforms and versions. If implementation details have to be tested, try to factor them out into separate test cases. Later on these test cases may be rewritten, or just skipped.

Also, aim for testing everything once, no less, no more. It's not effective having every test case fail just because one function in the interface changed.

2 Reference Manual

Common Test is a portable application for automated testing. It is suitable for black-box testing of target systems of any type (i.e. not necessarily implemented in Erlang), as well as for white-box testing of Erlang/OTP programs. Black-box testing is performed via standard O&M interfaces (such as SNMP, HTTP, Corba, Telnet, etc) and, if required, via user specific interfaces (often called test ports). White-box testing of Erlang/OTP programs is easily accomplished by calling the target API functions directly from the test case functions. Common Test also integrates usage of the OTP cover tool for code coverage analysis of Erlang/OTP programs.

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

Common Test provides libraries that contain useful support functions to fill various testing needs and requirements. There is for example support for flexible test declarations by means of so called test specifications. There is also support for central configuration and control of multiple independent test sessions (towards different target systems) running in parallel.

Common Test is implemented as a framework based on the OTP Test Server application.

common_test

Erlang module

The *Common Test* framework is an environment for implementing and performing automatic and semi-automatic execution of test cases. Common Test uses the OTP Test Server as engine for test case execution and logging.

In brief, Common Test supports:

- Automated execution of test suites (sets of test cases).
- Logging of the 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 sections describe the mandatory and optional test suite functions Common Test will call during test execution. For more details see *Common Test 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 | {group,GroupName} | {group,GroupName,Properties} |
{group,GroupName,Properties,SubGroups}]
TestCase = atom()
GroupName = atom()
Properties = [parallel | sequence | Shuffle | {RepeatType,N}] | default
SubGroups = [{GroupName,Properties} | {GroupName,Properties,SubGroups}]
Shuffle = shuffle | {shuffle,Seed}
Seed = {integer(),integer(),integer()}
RepeatType = repeat | repeat_until_all_ok | repeat_until_all_fail |
repeat_until_any_ok | repeat_until_any_fail
N = integer() | forever
Reason = term()
```

MANDATORY

This function must return the list of all test cases and test case groups in the test suite module that are to be executed. This list also specifies the order the cases and groups will be executed by Common Test. A test case is represented by an atom, the name of the test case function. 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 may also be specified, both for a top level group and for any of its sub-groups. Group execution properties specified here, will override properties in the group definition (see *`groups/0`*). (With value `default`, the group definition properties will be used).

If `{skip,Reason}` is returned, all test cases in the module will be skipped, and the `Reason` will be printed on the HTML result page.

For details on groups, see *Test case groups* in the User's Guide.

Module:groups() -> GroupDefs

Types:

```
GroupDefs = [Group]
Group = {GroupName,Properties,GroupsAndTestCases}
GroupName = atom()
Properties = [parallel | sequence | Shuffle | {RepeatType,N}]
GroupsAndTestCases = [Group | {group,GroupName} | TestCase]
TestCase = atom()
Shuffle = shuffle | {shuffle,Seed}
Seed = {integer(),integer(),integer()}
RepeatType = repeat | repeat_until_all_ok | repeat_until_all_fail |
repeat_until_any_ok | repeat_until_any_fail
N = integer() | forever
```

OPTIONAL

Function for defining test case groups. Please see *Test case groups* in the User's Guide for details.

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

This is the test suite info function. It is supposed to return a list of tagged tuples that specify various properties related to the execution of this test suite (common for all test cases in the suite).

The `timetrap` tag sets the maximum time 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 may also be used to trigger a timetrap timeout by, at some point, returning a value other than a `TimeVal`. (See the *User's Guide* for details).

The `require` tag specifies configuration variables that are required by test cases (and/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 more information about the 'require' functionality, see the reference manual for the function *`ct:require/1/2`*.

With `userdata`, it is possible for the user to specify arbitrary test suite related information which can be read by calling *`ct:userdata/2`*.

The `ct_hooks` tag specifies which *Common Test Hooks* are to be run together with this suite.

Other tuples than the ones defined will simply be ignored.

For more information about the test suite info function, see *Test suite info 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

This configuration function is called as the first function in the suite. It typically contains initializations which are common for all test cases in the suite, and which shall only be done once. The `Config` parameter is the configuration data which can be modified here. Whatever is returned from this function is given as `Config` to all configuration functions and test cases in the suite. If `{skip,Reason}` is returned, all test cases in the suite will be skipped and `Reason` printed in the overview log for the suite.

For information on `save_config` and `skip_and_save`, please see *Dependencies between Test Cases and Suites* in the User's Guide.

`Module:end_per_suite(Config) -> void() | {save_config,SaveConfig}`

Types:

`Config = SaveConfig = [{Key,Value}]`

`Key = atom()`

`Value = term()`

OPTIONAL

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`, please see *Dependencies between Test Cases and Suites* 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

This is the test case group info function. It is supposed to return a list of tagged tuples that specify various properties related to the execution of a test case group (i.e. its test cases and sub-groups). Properties set by *`group/1`* override properties with the same key that have been previously set by *`suite/0`*.

The `timetrap` tag sets the maximum time 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 may also be used to trigger a timetrap timeout by, at some point, returning a value other than a `TimeVal`. (See the *User's Guide* for details).

The `require` tag specifies configuration variables that are required by test cases (and/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 more information about the 'require' functionality, see the reference manual for the function *`ct:require/1/2`*.

With `userdata`, it is possible for the user to specify arbitrary test case group related information which can be read by calling *`ct:userdata/2`*.

The `ct_hooks` tag specifies which *Common Test Hooks* are to be run together with this suite.

Other tuples than the ones defined will simply be ignored.

For more information about the test case group info function, see *Test case group info 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

This configuration function is called before execution of a test case group. It typically contains initializations which are common for all test cases and sub-groups in the group, and which shall only be performed once. `GroupName` is the name of the group, as specified in the group definition (see *`groups/0`*). The `Config` parameter is the configuration data which can be modified here. The return value of this function is given as `Config` to all test cases and sub-groups in the group. If `{skip,Reason}` is returned, all test cases in the group will be skipped and `Reason` printed in the overview log for the group.

For information about test case groups, please see *Test case groups* chapter in the User's Guide.

`Module:end_per_group(GroupName, Config) -> void() | {return_group_result,Status}`

Types:

```
GroupName = atom()
Config = [{Key,Value}]
Key = atom()
Value = term()
Status = ok | skipped | failed
```

OPTIONAL

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`*. By means of `{return_group_result,Status}`, it is possible to return a status value for a nested sub-group. The status can be retrieved in *`end_per_group/2`* for the group on the level above. The status will also be used by Common Test for deciding if execution of a group should proceed in case the property `sequence` or `repeat_until_*` is set.

For more information about test case groups, please see *Test case groups* chapter 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

This function is called before each test case. The `TestCase` argument is the name of the test case, and `Config` (list of key-value tuples) is the configuration data that can be modified here. 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 will be skipped and `Reason` printed in the overview log for the suite.

`Module:end_per_testcase(TestCase, Config) -> void() | {fail,Reason} | {save_config,SaveConfig}`

Types:

```
TestCase = atom()
Config = SaveConfig = [{Key,Value}]
Key = atom()
Value = term()
Reason = term()
```

OPTIONAL

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` will be marked as failed (even though it was actually successful in the sense that it returned a value instead of terminating). For information on `save_config`, please see *Dependencies between Test Cases and Suites* 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

This is the test case info 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 that have been previously set for the test case by *`group/1`* or *`suite/0`*.

The `timetrap` tag sets the maximum time 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 may also be used to trigger a timetrap timeout by, at some point, returning a value other than a `TimeVal`. (See the *User's Guide* for details).

The `require` tag specifies configuration variables that are required by the test case (and/or `init/end_per_testcase/2`). If the required configuration variables are not found in any of the configuration files, the test case is skipped. For more information about the 'require' functionality, see the reference manual for the function *`ct:require/1/2`*.

If `timetrap` and/or `require` is not set, the default values specified by *`suite/0`* (or *`group/1`*) will be used.

With `userdata`, it is possible for the user to specify arbitrary test case related information which can be read by calling *`ct:userdata/3`*.

Other tuples than the ones defined will simply be ignored.

For more information about the test case info function, see *Test case info function* in the User's Guide.

`Module:Testcase(Config) -> void() | {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

This is the implementation of a test case. Here you must call the functions you want to test, and do whatever you need to check the result. If something fails, make sure 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 e.g. be read with `proplists:get_value/2` (or the macro `?config` defined in `ct.hrl`).

You can return `{skip,Reason}` if you decide not to run the test case after all. `Reason` will then be printed in 'Comment' field on the HTML result page.

You can return `{comment,Comment}` if you wish to print some information in the 'Comment' field on the HTML result page.

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 more information about test case implementation, please see *Test cases* in the User's Guide.

For information on `save_config` and `skip_and_save`, please see *Dependencies between Test Cases and Suites* in the User's Guide.

ct_run

Command

The `ct_run` program is automatically installed with Erlang/OTP and Common Test (please see the Installation chapter in the Common Test User's Guide for more information). The program accepts a number of different start flags. Some flags trigger `ct_run` to start the Common Test application and pass on data to it. Some flags start an Erlang node prepared for running Common Test in a particular mode.

There is an interface function that corresponds to this program, called *`ct:run_test/1`*, for starting Common Test from the Erlang shell (or an Erlang program). Please see the `ct` man page for details.

`ct_run` also accepts Erlang emulator flags. These are used when `ct_run` calls `erl` to start the Erlang node (making it possible to e.g. add directories to the code server path, change the cookie on the node, start additional applications, etc).

With the optional flag:

```
-erl_args
```

it's possible to divide the options on the `ct_run` command line into two groups, one that Common Test should process (those preceding `-erl_args`), and one it should completely ignore and pass on directly to the emulator (those following `-erl_args`). Options preceding `-erl_args` that Common Test doesn't recognize, also get passed on to the emulator untouched. By means of `-erl_args` the user may specify flags with the same name, but with different destinations, on the `ct_run` command line.

If `-pa` or `-pz` flags are specified in the Common Test group of options (preceding `-erl_args`), relative directories will be converted to absolute and re-inserted into the code path by Common Test (to avoid problems loading user modules when Common Test changes working directory during test runs). Common Test will however ignore `-pa` and `-pz` flags following `-erl_args` on the command line. These directories are added to the code path normally (i.e. 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]
    [-ct_hooks CTHModule1 CTHOpts1 and CTHModule2 CTHOpts2 and ..
 CTHModuleN CTHOptsN]
[-exit_status ignore_config]
```

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]
    [-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]
```

Refresh the HTML index files

```
ct_run -refresh_logs [-logdir LogDir] [-basic_html]
```

Run CT 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

Please read the *Running Test Suites* chapter in the Common Test User's Guide for information about the meaning of the different start flags.

ct

Erlang module

Main user interface for the Common Test framework.

This module implements the command line interface for running tests and some 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 should be used to retrieve information from the `Config` variable sent to all test cases. It is used with two arguments, where the first is the name of the configuration variable you wish to retrieve, and 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() = handle() (see module ct_gen_conn) | term()`

The identity of a specific connection.

`target_name() = var_name()`

The name of a target.

`var_name() = atom()`

A variable name which is specified when `ct:require/2` is called, e.g. `ct:require(mynodename, {node,[telnet]})`

Exports

`abort_current_testcase(Reason) -> ok | {error, ErrorReason}`

Types:

`Reason = term()`
`ErrorReason = no_testcase_running | parallel_group`

When calling this function, the currently executing test case will be aborted. It is the user's responsibility to know for sure which test case is currently executing. The function is therefore only safe to call from a function which 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()`

This function loads configuration variables using the given callback module and configuration string. Callback module should be either loaded or present in the code part. Loaded configuration variables can later be removed using `remove_config/2` function.

`break(Comment) -> ok | {error, Reason}`

Types:

```
Comment = string()
Reason = {multiple_cases_running, TestCases} | 'enable break with
release_shell option'
TestCases = [atom()]
```

This function will cancel any active timetrap and pause the execution of the current test case until the user calls the `continue/0` function. It gives the user the opportunity to interact with the erlang node running the tests, e.g. for debugging purposes or for manually executing a part of the test case. If a parallel group is executing, `break/2` should be called instead.

A cancelled timetrap will not be 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 the `release_shell` start option to `true`. See the User's Guide for more information.

`break(TestCase, Comment) -> ok | {error, Reason}`

Types:

```
TestCase = atom()
Comment = string()
Reason = 'test case not running' | 'enable break with release_shell
option'
```

This function works the same way as `break/1`, only the `TestCase` argument makes it possible to pause a test case executing in a parallel group. The `continue/1` function should be used to resume execution of `TestCase`.

See `break/1` for more details.

`capture_get() -> ListOfStrings`

Types:

```
ListOfStrings = [string()]
```

Equivalent to *capture_get([default])*.

`capture_get(ExclCategories) -> ListOfStrings`

Types:

```
ExclCategories = [atom()]
ListOfStrings = [string()]
```

Return and purge the list of text strings buffered during the latest session of capturing printouts to stdout. With `ExclCategories` it's possible to specify log categories that should be ignored in `ListOfStrings`. If `ExclCategories = []`, no filtering takes place.

See also: capture_start/0, capture_stop/0, log/3.

capture_start() -> ok

Start capturing all text strings printed to stdout during execution of the test case.

See also: capture_get/1, capture_stop/0.

capture_stop() -> ok

Stop capturing text strings (a session started with `capture_start/0`).

See also: capture_get/1, capture_start/0.

comment(Comment) -> void()

Types:

Comment = term()

Print the given `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) -> void()

Types:

Format = string()

Args = list()

Print the formatted string in the comment field in the table on the test suite result page.

The `Format` and `Args` arguments are used in call to `io_lib:format/2` in order to create the comment string. The behaviour of `comment/2` is otherwise the same as the `comment/1` function (see above for details).

continue() -> ok

This function must be called in order to continue after a test case (not executing in a parallel group) has called `break/1`.

continue(TestCase) -> ok

Types:

TestCase = atom()

This function must be called in order to continue after a test case has called `break/2`. If the paused test case, `TestCase`, executes in a parallel group, this function - rather than `continue/0` - must be used in order to let the test case proceed.

decrypt_config_file(EncryptFileName, TargetFileName) -> ok | {error, Reason}

Types:

EncryptFileName = string()

TargetFileName = string()

Reason = term()

This function decrypts `EncryptFileName`, previously generated with `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` 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()

This function decrypts `EncryptFileName`, previously generated with `encrypt_config_file/2/3`. The original file contents is saved in the target file. The key must have the the same value as that used for encryption.

encrypt_config_file(SrcFileName, EncryptFileName) -> ok | {error, Reason}

Types:

SrcFileName = string()
EncryptFileName = string()
Reason = term()

This function encrypts the source config 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` in the current directory, or the home directory of the user (it is searched for in that order).

See the Common Test User's Guide for information about using encrypted config files when running tests.

See the `crypto` application for details on DES3 encryption/decryption.

encrypt_config_file(SrcFileName, EncryptFileName, KeyOrFile) -> ok | {error, Reason}

Types:

SrcFileName = string()
EncryptFileName = string()
KeyOrFile = {key, string()} | {file, string()}
Reason = term()

This function encrypts the source config 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}`.

See the Common Test User's Guide for information about using encrypted config files when running tests.

See the `crypto` application for details on DES3 encryption/decryption.

fail(Reason) -> void()

Types:

Reason = term()

Terminate a test case with the given error `Reason`.

fail(Format, Args) -> void()

Types:

Format = string()
Args = list()

Terminate 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 *get_config(Required, undefined, [])*.

`get_config(Required, Default) -> Value`

Equivalent to *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
```

Read config data values.

This function returns the matching value(s) or config element(s), given a config variable key or its associated name (if one has been specified with `require/2` or a require statement).

Example, given the following config file:

```
{unix,[{telnet,IpAddr},
       {user,[{username,Username},
              {password,Password}]}]}.
```

```
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 config variable key has been associated with a name (by means of `require/2` or a require statement), the name may 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 config variable is defined in multiple files and you want to access all possible values, use the `all` option. The values will be returned in a list and the order of the elements corresponds to the order that the config files were specified at startup.

If you want config elements (key-value tuples) returned as result instead of values, use the `element` option. The returned elements will then be on the form `{Required,Value}`

See also: get_config/1, *get_config/2*, *require/1*, *require/2*.

get_event_mgr_ref() -> EvMgrRef

Types:

EvMgrRef = atom()

Call this function in order to get a reference to the CT event manager. The reference can be used to e.g. add a user specific event handler while tests are running. Example: `gen_event:add_handler(ct:get_event_mgr_ref(), my_ev_h, [])`

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 info about which test case is currently 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()

Return the name of the target that the given connection belongs to.

get_timetrap_info() -> {Time, Scale}

Types:

Time = integer() | infinity

Scale = true | false

Read info about the timetrap set for the current test case. `Scale` indicates if Common Test will attempt to automatically compensate timetraps for runtime delays introduced by e.g. tools like cover.

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()
```

Install config files and event handlers.

Run this function once before first test.

Example:
`install([{config,["config_node.ctc","config_user.ctc"]}])`.

Note that this function is automatically run by the `ct_run` program.

`listenv(Telnet) -> [Env]`

Types:

```
Telnet = term()
Env = {Key, Value}
Key = string()
Value = string()
```

Performs the listenv command on the given telnet connection and returns the result as a list of Key-Value pairs.

`log(Format) -> ok`

Equivalent to *log(default, 50, Format, [])*.

`log(X1, X2) -> ok`

Types:

```
X1 = Category | Importance | Format
X2 = Format | Args
```

Equivalent to *log(Category, Importance, Format, Args)*.

`log(X1, X2, X3) -> ok`

Types:

```
X1 = Category | Importance
X2 = Importance | Format
X3 = Format | Args
```

Equivalent to *log(Category, Importance, Format, Args)*.

`log(Category, Importance, Format, Args) -> ok`

Types:

```
Category = atom()
Importance = integer()
Format = string()
Args = list()
```

Printout 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 `Args` is `[]`.

Please see the User's Guide for details on `Category` and `Importance`.

`make_priv_dir() -> ok | {error, Reason}`

Types:

`Reason = term()`

If the test has been started with the create_priv_dir option 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 a 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 *pal(default, 50, Format, [])*.

`pal(X1, X2) -> ok`

Types:

`X1 = Category | Importance | Format`
`X2 = Format | Args`

Equivalent to *pal(Category, Importance, Format, Args)*.

`pal(X1, X2, X3) -> ok`

Types:

`X1 = Category | Importance`
`X2 = Importance | Format`
`X3 = Format | Args`

Equivalent to *pal(Category, Importance, Format, Args)*.

`pal(Category, Importance, Format, Args) -> ok`

Types:

`Category = atom()`
`Importance = integer()`
`Format = string()`
`Args = list()`

Print and log 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 `Args` is `[]`.

Please see the User's Guide for details on `Category` and `Importance`.

`parse_table(Data) -> {Heading, Table}`

Types:

```
Data = [string()]
Heading = tuple()
Table = [tuple()]
```

Parse the printout from an SQL table and return a list of tuples.

The printout to parse would typically be 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 *print(default, 50, Format, [])*.

`print(X1, X2) -> ok`

Types:

```
X1 = Category | Importance | Format
X2 = Format | Args
```

Equivalent to *print(Category, Importance, Format, Args)*.

`print(X1, X2, X3) -> ok`

Types:

```
X1 = Category | Importance
X2 = Importance | Format
X3 = Format | Args
```

Equivalent to *print(Category, Importance, Format, Args)*.

`print(Category, Importance, Format, Args) -> ok`

Types:

```
Category = atom()
Importance = integer()
Format = string()
Args = list()
```

Printout 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 `Args` is `[]`.

Please see the User's Guide for details on `Category` and `Importance`.

reload_config(Required) -> ValueOrElement

Types:

Required = KeyOrName | {KeyOrName, SubKey} | {KeyOrName, SubKey, SubKey}
KeyOrName = atom()
SubKey = atom()
ValueOrElement = term()

Reload config file which contains specified configuration key.

This function performs updating of the configuration data from which the given configuration variable was read, and returns the (possibly) new value of this variable.

Note that if some variables were present in the configuration but are not loaded using this function, they will be removed from the configuration table together with their aliases.

remove_config(Callback, Config) -> ok

Types:

Callback = atom()
Config = string()
Reason = term()

This function removes configuration variables (together with their aliases) which 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()

Check if the required configuration is available. It is possible to specify arbitrarily deep tuples as `Required`. Note that it is only the last element of the tuple which can be a list of `SubKey`s.

Example 1: require the variable `myvar`:

```
ok = ct:require(myvar).
```

In this case the config file must at least contain:

```
{myvar,Value}.
```

Example 2: require the key `myvar` with subkeys `sub1` and `sub2`:

```
ok = ct:require({myvar,[sub1,sub2]}).
```

In this case the config file must at least contain:

```
{myvar,[{sub1,Value},{sub2,Value}]}.
```

Example 3: require the key `myvar` with subkey `sub1` with `subsub1`:

```
ok = ct:require({myvar,sub1,sub2}).
```

In this case the config file must at least contain:

```
{myvar,[{sub1,[{sub2,Value}]}]}.
```

See also: get_config/1, get_config/2, get_config/3, require/2.

`require(Name, Required) -> ok | {error, Reason}`

Types:

`Name = atom()`
`Required = Key | {Key, SubKey} | {Key, SubKey, SubKey}`
`SubKey = Key`
`Key = atom()`

Check if the required configuration is available, and give it a name. The semantics for `Required` is the same as in `required/1` except that it is not possible to specify a list of `SubKey`s.

If the requested data is available, the sub entry will be associated with `Name` so that the value of the element can be read with `get_config/1,2` provided `Name` 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 may then use the node name. E.g. you can fetch a file over ftp like this:

```
ok = ct:ftp_get(a,RemoteFile,LocalFile).
```

For this to work, the config file must at least contain:

```
{machine,[{node,[{telnet,IpAddr},{ftp,IpAddr}]}]}.
```

> Note:
>
> The behaviour of this function changed radically in common_test 1.6.2. In order too keep some backwards compatability it is still possible to do:
> `ct:require(a,{node,[telnet,ftp]}).`
> This will associate the name `a` with the top level `node` entry. For this to work, the config file must at least contain:
> `{node,[{telnet,IpAddr},{ftp,IpAddr}]}.`

See also: get_config/1, get_config/2, get_config/3, require/1.

`run(TestDirs) -> Result`

Types:

```
TestDirs = TestDir | [TestDir]
```

Run all test cases in all suites in the given directories.

See also: run/3.

`run(TestDir, Suite) -> Result`

Run all test cases in the given suite.

See also: run/3.

`run(TestDir, Suite, Cases) -> Result`

Types:

```
TestDir = string()
Suite = atom()
Cases = atom() | [atom()]
Result = [TestResult] | {error, Reason}
```

Run the given test case(s).

Requires that `ct:install/1` has been run first.

Suites (*_SUITE.erl) files must be stored in `TestDir` or `TestDir/test`. All suites will be compiled when 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} |
{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()
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()
```

Run tests as specified by the combination of options in `Opts`. The options are the same as those used with the *`ct_run`* program. Note that here a `TestDir` can be used to point out the path to a `Suite`. Note also that the option `testcase` corresponds to the `-case` option in the `ct_run` program. Configuration files specified in `Opts` will be installed automatically at startup.

`TestRunnerPid` is returned if `release_shell == true` (see `break/1` for details).

`Reason` indicates what type of error has been 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()
```

Run test specified by `TestSpec`. The terms are the same as those used in test specification files.

`Reason` indicates what type of error has been encountered.

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`, suspends the test case for 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

Start CT 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 using "required config data" (e.g. telnet or ftp functions) are to be called from the erlang shell, config 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()
```

Step through a test case with the debugger.

See also: run/3.

step(TestDir, Suite, Case, Opts) -> Result

Types:

Case = atom()
Opts = [Opt] | []
Opt = config | keep_inactive

Step through a test case with the debugger. If the `config` option has been given, breakpoints will be set also on the configuration functions in `Suite`.

See also: run/3.

stop_interactive() -> ok

Exit the interactive mode.

See also: 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() | float()
Func = {M, F, A} | function()
M = atom()
F = atom()
A = list()

Use this function to set a new timetrap for the running test case. If the argument is `Func`, the timetrap will be triggered when this function returns. `Func` may also return a new `Time` value, which in that case will be 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 the tag `userdata` in the list of tuples returned from `Suite: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 the 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 will be written on a primitive html format, not using the Common Test CSS style sheet.

`get_event_mgr_ref() -> MasterEvMgrRef`

Types:

```
MasterEvMgrRef = atom()
```

Call this function in order to get a reference to the CT master event manager. The reference can be used to e.g. 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 have not yet finished.

`run(TestSpecs) -> ok`

Types:

```
TestSpecs = string() | [SeparateOrMerged]
```

Equivalent to *run(TestSpecs, false, [], [])*.

`run(TestSpecs, InclNodes, ExclNodes) -> ok`

Types:

```
TestSpecs = string() | [SeparateOrMerged]
SeparateOrMerged = string() | [string()]
InclNodes = [atom()]
ExclNodes = [atom()]
```

Equivalent to *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 will be handled separately. It is however possible to also specify a list of specifications that should be merged into one before the tests are executed. Any test without a particular node specification will also be executed on the nodes in `InclNodes`. Nodes in the `ExclNodes` list will be excluded from the test.

run_on_node(TestSpecs, Node) -> ok

Types:

```
TestSpecs = string() | [SeparateOrMerged]
SeparateOrMerged = string() | [string()]
Node = atom()
```

Equivalent to *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
```

Add nodes to current cover test (only works if cover support is active!). To have effect, this function should be called from init_per_suite/1 before any actual tests are performed.

`cross_cover_analyse(Level, Tests) -> ok`

Types:

```
Level = overview | details
Tests = [{Tag, Dir}]
Tag = atom()
Dir = string()
```

Accumulate cover results over multiple tests. See the chapter about *cross cover analysis* in the users's guide.

`remove_nodes(Nodes) -> ok | {error, Reason}`

Types:

```
Nodes = [atom()]
Reason = cover_not_running | not_main_node
```

Remove nodes from current cover test. Call this function to stop cover test on nodes previously added with 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 support of the INETS application).

DATA TYPES

```
connection() = handle() | target_name() (see module ct)
handle() = handle() (see module ct_gen_conn)
```

Handle for a specific ftp connection.

Exports

`cd(Connection, Dir) -> ok | {error, Reason}`

Types:

`Connection = connection()`
`Dir = string()`

Change directory on remote host.

`close(Connection) -> ok | {error, Reason}`

Types:

`Connection = connection()`

Close the FTP connection.

`delete(Connection, File) -> ok | {error, Reason}`

Types:

`Connection = connection()`
`File = string()`

Delete a file on remote host

`get(KeyOrName, RemoteFile, LocalFile) -> ok | {error, Reason}`

Types:

`KeyOrName = Key | Name`
`Key = atom()`
`Name = target_name() (see module ct)`
`RemoteFile = string()`
`LocalFile = string()`

Open a ftp connection and fetch a file from the remote host.

`RemoteFile` and `LocalFile` must be absolute paths.

The config file must be as for put/3.

See also: put/3, ct:require/2.

ls(Connection, Dir) -> {ok, Listing} | {error, Reason}

Types:

Connection = connection()
Dir = string()
Listing = string()

List the directory Dir.

open(KeyOrName) -> {ok, Handle} | {error, Reason}

Types:

KeyOrName = Key | Name
Key = atom()
Name = target_name() (see module ct)
Handle = handle()

Open an FTP connection to the specified node.

You can open one connection for a particular `Name` and use the same name as reference for all subsequent operations. If you want the connection to be associated with `Handle` instead (in case you need to open multiple connections to a host for example), simply use `Key`, the configuration variable name, to specify the target. Note that a connection that has no associated target name can only be closed with the handle value.

See `ct:require/2` for how to create a new `Name`

See also: ct:require/2.

put(KeyOrName, LocalFile, RemoteFile) -> ok | {error, Reason}

Types:

KeyOrName = Key | Name
Key = atom()
Name = target_name() (see module ct)
LocalFile = string()
RemoteFile = string()

Open a ftp connection and send a file to the remote host.

`LocalFile` and `RemoteFile` must be absolute paths.

If the target host is a "special" node, the ftp address must be specified in the config file like this:

```
{node,[{ftp,IpAddr}]}.
```

If the target host is something else, e.g. a unix host, the config 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}

Fetch a file over FTP.

The file will get the same name on the local host.

See also: recv/3.

recv(Connection, RemoteFile, LocalFile) -> ok | {error, Reason}

Types:

```
Connection = connection()
RemoteFile = string()
LocalFile = string()
```

Fetch a file over FTP.

The file will be named `LocalFile` on the local host.

send(Connection, LocalFile) -> ok | {error, Reason}

Send a file over FTP.

The file will get the same name on the remote host.

See also: send/3.

send(Connection, LocalFile, RemoteFile) -> ok | {error, Reason}

Types:

```
Connection = connection()
LocalFile = string()
RemoteFile = string()
```

Send a file over FTP.

The file will be named `RemoteFile` on the remote host.

type(Connection, Type) -> ok | {error, Reason}

Types:

```
Connection = connection()
Type = ascii | binary
```

Change file transfer type

ct_ssh

Erlang module

SSH/SFTP client module.

ct_ssh uses the OTP ssh application and more detailed information about e.g. functions, types and options can be found in the documentation for this application.

The `Server` argument in the SFTP functions should only be used for SFTP sessions that have been started on existing SSH connections (i.e. 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 (i.e. may be used as config 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`.

Please see ssh(3) for other types.

All timeout parameters in ct_ssh functions are values in milliseconds.

DATA TYPES

`connection() = handle() | target_name() (see module ct)`
`handle() = handle() (see module ct_gen_conn)`

Handle for a specific SSH/SFTP connection.

`ssh_sftp_return() = term()`

A 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 info 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 info 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 info 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 info and other types, see ssh_sftp(3).

aread(SSH, Handle, Len) -> Result

Types:

```
SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()
```

For info 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 info and other types, see ssh_sftp(3).

awrite(SSH, Handle, Data) -> Result

Types:

```
SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()
```

For info 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 info and other types, see ssh_sftp(3).

close(SSH, Handle) -> Result

Types:

```
SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()
```

For info and other types, see ssh_sftp(3).

close(SSH, Server, Handle) -> Result

Types:

```
SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()
```

For info and other types, see ssh_sftp(3).

connect(KeyOrName) -> {ok, Handle} | {error, Reason}

Equivalent to *connect(KeyOrName, host, [])*.

connect(KeyOrName, ConnType) -> {ok, Handle} | {error, Reason}

Equivalent to *connect(KeyOrName, ConnType, [])*.

connect(KeyOrName, ConnType, ExtraOpts) -> {ok, Handle} | {error, Reason}

Types:

```
KeyOrName = Key | Name
Key = atom()
Name = target_name() (see module ct)
ConnType = ssh | sftp | host
ExtraOpts = ssh_connect_options()
Handle = handle()
Reason = term()
```

Open 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 may be used as connection reference for subsequent calls. It's only possible to have one open connection at a time associated with `Name`. If `Key` is used, the returned handle must be used for subsequent calls (multiple connections may be opened using the config data specified by `Key`). See `ct:require/2` for how to create a new `Name`

`ConnType` will always override 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. The value `host` means the connection type specified by the host option (either in the configuration data or in `ExtraOpts`) will be used.

`ExtraOpts` (optional) are extra SSH options to be added to the config data for `KeyOrName`. The extra options will override any existing options with the same key in the config data. For details on valid SSH options, see the documentation for the OTP ssh application.

See also: ct:require/2.

`del_dir(SSH, Name) -> Result`

Types:

```
SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()
```

For info 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 info and other types, see ssh_sftp(3).

`delete(SSH, Name) -> Result`

Types:

```
SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()
```

For info and other types, see ssh_sftp(3).

`delete(SSH, Server, Name) -> Result`

Types:

```
SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()
```

For info and other types, see ssh_sftp(3).

`disconnect(SSH) -> ok | {error, Reason}`

Types:

```
SSH = connection()
Reason = term()
```

Close an SSH/SFTP connection.

`exec(SSH, Command) -> {ok, Data} | {error, Reason}`

Equivalent to *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 info 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 info and other types, see ssh_sftp(3).

list_dir(SSH, Path) -> Result

Types:

```
SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()
```

For info 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 info and other types, see ssh_sftp(3).

make_dir(SSH, Name) -> Result

Types:

```
SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()
```

For info 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 info 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 info 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 info and other types, see ssh_sftp(3).

open(SSH, File, Mode) -> Result

Types:

```
SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()
```

For info 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 info and other types, see ssh_sftp(3).

opendir(SSH, Path) -> Result

Types:

```
SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()
```

For info and other types, see ssh_sftp(3).

opendir(SSH, Server, Path) -> Result

Types:

```
SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()
```

For info and other types, see ssh_sftp(3).

position(SSH, Handle, Location) -> Result

Types:

```
SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()
```

For info 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 info 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 info 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 info 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 info 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 info and other types, see ssh_sftp(3).

read(SSH, Handle, Len) -> Result

Types:

```
SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()
```

For info 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 info and other types, see ssh_sftp(3).

read_file(SSH, File) -> Result

Types:

```
SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()
```

For info 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 info 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 info 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 info and other types, see ssh_sftp(3).

read_link(SSH, Name) -> Result

Types:

```
SSH = connection()
Result = ssh_sftp_return() | {error, Reason}
Reason = term()
```

For info 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 info 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 info 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 info and other types, see ssh_sftp(3).

receive_response(SSH, ChannelId) -> {ok, Data} | {error, Reason}

Equivalent to *receive_response(SSH, ChannelId, close)*.

receive_response(SSH, ChannelId, End) -> {ok, Data} | {error, Reason}

Equivalent to *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 timeout occurs before this happens, the function returns `{timeout,Data}` (where `Data` is the data received so far). If `End == timeout`, a timeout is expected and `{ok,Data}` is returned both in the case of a timeout and when the channel is closed. If `End` is a fun, this fun will be called with one argument - the data value in a received `ssh_cm` message (see ssh_connection(3)). The fun should return `true` to end the receiving operation (and have the so far collected data returned), or `false` to wait for more data from the server. (Note that 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 info 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 info and other types, see ssh_sftp(3).

send(SSH, ChannelId, Data) -> ok | {error, Reason}

Equivalent to *send(SSH, ChannelId, 0, Data, DefaultTimeout)*.

send(SSH, ChannelId, Data, Timeout) -> ok | {error, Reason}

Equivalent to *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()
```

Send data to server on specified session channel.

send_and_receive(SSH, ChannelId, Data) -> {ok, Data} | {error, Reason}

Equivalent to *send_and_receive(SSH, ChannelId, Data, close)*.

send_and_receive(SSH, ChannelId, Data, End) -> {ok, Data} | {error, Reason}

Equivalent to *send_and_receive(SSH, ChannelId, 0, Data, End, DefaultTimeout)*.

send_and_receive(SSH, ChannelId, Data, End, Timeout) -> {ok, Data} | {error, Reason}

Equivalent to *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()
```

Send data to server on specified session channel and wait to receive the server response.

See `receive_response/4` for details on the `End` argument.

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 *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.

`subsystem(SSH, ChannelId, Subsystem) -> Status | {error, Reason}`

Equivalent to *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 info 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 info 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 info 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 info 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 info 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 info and other types, see ssh_sftp(3).

ct_netconfc

Erlang module

Netconf client module.

The Netconf client is compliant with RFC4741 and RFC4742.

For each server to test against, the following entry can be added to a configuration file:

`{server_id(),options()}.`

The `server_id()` or an associated `target_name()` (see *ct*) shall then be used in calls to *open/2*.

If no configuration exists for a server, a session can still be opened by calling *open/2* with all necessary options given in the call. The first argument to *open/2* can then be any atom.

Logging

The netconf server uses the `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 your test suite, e.g.

```
suite() ->
   [{ct_hooks, [{cth_conn_log, [{conn_mod(),hook_options()}]}]}].
```

The `conn_mod()` is the name of the common_test module implementing the connection protocol, e.g. `ct_netconfc`.

The 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, it is possible to have different connections logged in separate files. To do this, use the hook option `hosts` and list the names of the servers/connections that will be used in the suite. Note that the connections must be named for this to work, i.e. they must be opened with *open/2*.

The `hosts` option has no effect if `log_type` is set to `html` or `silent`.

The hook options can also be specified in a configuration file with the configuration variable `ct_conn_log`:

```
{ct_conn_log,[{conn_mod(),hook_options()}]}.
```

For example:

```
{ct_conn_log,[{ct_netconfc,[{log_type,pretty},
```

```
                                {hosts,[key_or_name()]}]}]}
```

Note that hook options specified in a configuration file will overwrite the hardcoded hook options in the test suite.

Logging example 1

The following `ct_hooks` statement will cause pretty printing of netconf traffic to separate logs for the connections named `nc_server1` and `nc_server2`. Any other connections will be 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 like this:

```
open(nc_server1,[...]),
open(nc_server2,[...]).
```

Logging example 2

The following configuration file will cause raw logging of all netconf traffic into one single text file.

```
{ct_conn_log,[{ct_netconfc,[{log_type,raw}]}]}.
```

The `ct_hooks` statement must look like this:

```
suite() ->
   [{ct_hooks, [{cth_conn_log, []}]}].
```

The same `ct_hooks` statement without the configuration file would cause HTML logging of all netconf connections into the test case HTML log.

Notifications

The netconf client is also compliant with RFC5277 NETCONF Event Notifications, which defines a mechanism for an asynchronous message notification delivery service for the netconf protocol.

Specific functions to support this are *create_subscription/6* and *get_event_streams/3*. (The functions also exist with other arities.)

DATA TYPES

```
client() = handle() | server_id() (see module ct_gen_conn) | target_name()
(see module ct_gen_conn)
error_reason() = term()
event_time() = {eventTime, xml_attributes(), [xs_datetime()]}
handle() = term()
```

An opaque reference for a connection (netconf session). See *ct* for more information.

```
host() = hostname() (see module inet) | ip_address() (see module inet)
netconf_db() = running | startup | candidate
notification() = {notification, xml_attributes(), notification_content()}
notification_content() = [event_time() | simple_xml()]
```

```
option() = {ssh, host()} | {port, port_number() (see module inet)} |
{user, string()} | {password, string()} | {user_dir, string()} | {timeout,
timeout()}
options() = [option()]
```

Options used for setting up ssh connection to a netconf server.

```
simple_xml() = {xml_tag(), xml_attributes(), xml_content()} | {xml_tag(),
xml_content()} | xml_tag()
```

This type is further described in the documentation for the `Xmerl` application.

```
stream_data() = {description, string()} | {replaySupport, string()} |
{replayLogCreationTime, string()} | {replayLogAgedTime, string()}
```

See XML Schema for Event Notifications found in RFC5277 for further detail about the data format for the string values.

```
stream_name() = string()
streams() = [{stream_name(), [stream_data()]}]
xml_attribute_tag() = atom()
xml_attribute_value() = string()
xml_attributes() = [{xml_attribute_tag(), xml_attribute_value()}]
xml_content() = [simple_xml() | iolist()]
xml_tag() = atom()
xpath() = {xpath, string()}
xs_datetime() = string()
```

This date and time identifyer has the same format as the XML type dateTime and compliant to RFC3339. The format is

```
[-]CCYY-MM-DDThh:mm:ss[.s][Z|(+|-)hh:mm]
```

Exports

`action(Client, Action) -> Result`

Equivalent to *action(Client, Action, infinity)*.

`action(Client, Action, Timeout) -> Result`

Types:

`Client = client()`
`Action = simple_xml()`
`Timeout = timeout()`
`Result = ok | {ok, [simple_xml()]} | {error, error_reason()}`

Execute an action. If the return type is void, `ok` will be returned instead of `{ok,[simple_xml()]}`.

`close_session(Client) -> Result`

Equivalent to *close_session(Client, infinity)*.

`close_session(Client, Timeout) -> Result`

Types:

```
Client = client()
Timeout = timeout()
Result = ok | {error, error_reason()}
```

Request graceful termination of the session associated with the client.

When a netconf server receives a `close-session` request, it will gracefully close the session. The server will release any locks and resources associated with the session and gracefully close any associated connections. Any NETCONF requests received after a `close-session` request will be ignored.

`copy_config(Client, Source, Target) -> Result`

Equivalent to *copy_config(Client, Source, Target, infinity)*.

`copy_config(Client, Target, Source, Timeout) -> Result`

Types:

```
Client = client()
Target = netconf_db()
Source = netconf_db()
Timeout = timeout()
Result = ok | {error, error_reason()}
```

Copy configuration data.

Which source and target options that can be issued depends on the capabilities supported by the server. I.e. `:candidate` and/or `:startup` are required.

`create_subscription(Client) -> term()`

`create_subscription(Client, Timeout) -> term()`

`create_subscription(Client, Stream, Timeout) -> term()`

`create_subscription(Client, StartTime, StopTime, Timeout) -> term()`

`create_subscription(Client, Stream, StartTime, StopTime, Timeout) -> term()`

`create_subscription(Client, Stream, Filter, StartTime, StopTime, Timeout) -> Result`

Types:

```
Client = client()
Stream = stream_name()
Filter = simple_xml() | [simple_xml()]
StartTime = xs_datetime()
StopTime = xs_datetime()
Timeout = timeout()
Result = ok | {error, error_reason()}
```

Create a subscription for event notifications.

This function sets up a subscription for netconf event notifications of the given stream type, matching the given filter. The calling process will receive notifications as messages of type `notification()`.

Stream:

An optional parameter that indicates which stream of events is of interest. If not present, events in the default NETCONF stream will be sent.

Filter:

An optional parameter that indicates which subset of all possible events is of interest. The format of this parameter is the same as that of the filter parameter in the NETCONF protocol operations. If not present, all events not precluded by other parameters will be sent.

StartTime:

An optional parameter used to trigger the replay feature and indicate that the replay should 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 the `StartTime` specified is earlier than the log can support, the replay will begin with the earliest available notification. This parameter is of type dateTime and compliant to [RFC3339]. Implementations must support time zones.

StopTime:

An optional parameter used with the optional replay feature to indicate the newest notifications of interest. If `StopTime` is not present, the notifications will continue 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 [RFC3339]. Implementations must support time zones.

See RFC5277 for further details about the event notification mechanism.

`delete_config(Client, Target) -> Result`

Equivalent to *delete_config(Client, Target, infinity)*.

`delete_config(Client, Target, Timeout) -> Result`

Types:

```
Client = client()
Target = startup | candidate
Timeout = timeout()
Result = ok | {error, error_reason()}
```

Delete configuration data.

The running configuration cannot be deleted and `:candidate` or `:startup` must be advertised by the server.

`edit_config(Client, Target, Config) -> Result`

Equivalent to *edit_config(Client, Target, Config, infinity)*.

`edit_config(Client, Target, Config, Timeout) -> Result`

Types:

```
Client = client()
Target = netconf_db()
Config = simple_xml()
Timeout = timeout()
```

Result = ok | {error, error_reason()}

Edit configuration data.

Per default only the running target is available, unless the server include `:candidate` or `:startup` in its list of capabilities.

format_data(How, Data) -> term()

get(Client, Filter) -> Result

Equivalent to *get(Client, Filter, infinity)*.

get(Client, Filter, Timeout) -> Result

Types:

Client = client()
Filter = simple_xml() | xpath()
Timeout = timeout()
Result = {ok, [simple_xml()]} | {error, error_reason()}

Get data.

This operation returns both configuration and state data from the server.

Filter type `xpath` can only be used if the server supports `:xpath`.

get_capabilities(Client) -> Result

Equivalent to *get_capabilities(Client, infinity)*.

get_capabilities(Client, Timeout) -> Result

Types:

Client = client()
Timeout = timeout()
Result = [string()] | {error, error_reason()}

Returns the server side capabilities

The following capability identifiers, defined in RFC 4741, can be returned:

- `"urn:ietf:params:netconf:base:1.0"`
- `"urn:ietf:params:netconf:capability:writable-running:1.0"`
- `"urn:ietf:params:netconf:capability:candidate:1.0"`
- `"urn:ietf:params:netconf:capability:confirmed-commit:1.0"`
- `"urn:ietf:params:netconf:capability:rollback-on-error:1.0"`
- `"urn:ietf:params:netconf:capability:startup:1.0"`
- `"urn:ietf:params:netconf:capability:url:1.0"`
- `"urn:ietf:params:netconf:capability:xpath:1.0"`

Note, additional identifiers may exist, e.g. server side namespace.

get_config(Client, Source, Filter) -> Result

Equivalent to *get_config(Client, Source, Filter, infinity)*.

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()}

Get configuration data.

To be able to access another source than `running`, the server must advertise `:candidate` and/or `:startup`.

Filter type `xpath` can only be used if the server supports `:xpath`.

get_event_streams(Client, Timeout) -> Result

Equivalent to *get_event_streams(Client, [], Timeout)*.

get_event_streams(Client, Streams, Timeout) -> Result

Types:

Client = client()
Streams = [stream_name()]
Timeout = timeout()
Result = {ok, streams()} | {error, error_reason()}

Send a request to get the given event streams.

`Streams` is a list of stream names. The following filter will be 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 will be requested by sending the following filter:

```
<netconf xmlns="urn:ietf:params:xml:ns:netmod:notification">
  <streams/>
</netconf>
```

If more complex filtering is needed, a use *get/2* or *get/3* and specify the exact filter according to XML Schema for Event Notifications found in RFC5277.

get_session_id(Client) -> Result

Equivalent to *get_session_id(Client, infinity)*.

get_session_id(Client, Timeout) -> Result

Types:

```
Client = client()
Timeout = timeout()
Result = pos_integer() | {error, error_reason()}
```

Returns the session id associated with the given client.

handle_msg(X1, State) -> term()

hello(Client) -> Result

Equivalent to *hello(Client, [], infinity)*.

hello(Client, Timeout) -> Result

Equivalent to *hello(Client, [], Timeout)*.

hello(Client, Options, Timeout) -> Result

Types:

```
Client = handle()
Options = [{capability, [string()]}]
Timeout = timeout()
Result = ok | {error, error_reason()}
```

Exchange `hello` messages with the server.

Adds optional capabilities and sends a `hello` message to the server and waits for the return.

kill_session(Client, SessionId) -> Result

Equivalent to *kill_session(Client, SessionId, infinity)*.

kill_session(Client, SessionId, Timeout) -> Result

Types:

```
Client = client()
SessionId = pos_integer()
Timeout = timeout()
Result = ok | {error, error_reason()}
```

Force termination of the session associated with the supplied session id.

The server side shall abort any operations currently in process, 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 will be restored to its state before entering the confirmed commit phase. Otherwise, no configuration roll back will be performed.

If the given `SessionId` is equal to the current session id, an error will be returned.

lock(Client, Target) -> Result

Equivalent to *lock(Client, Target, infinity)*.

lock(Client, Target, Timeout) -> Result

Types:

Client = client()
Target = netconf_db()
Timeout = timeout()
Result = ok | {error, error_reason()}

Unlock 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 not available to other clients (Netconf, CORBA, SNMP etc). Locks are intended to be short-lived.

The operations *kill_session/2* or *kill_session/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 = options()
Result = {ok, handle()} | {error, error_reason()}

Open a netconf session, but don't send `hello`.

As *open/1* but does not send a `hello` message.

only_open(KeyOrName, ExtraOptions) -> Result

Types:

KeyOrName = key_or_name() (see module ct_gen_conn)
ExtraOptions = options()
Result = {ok, handle()} | {error, error_reason()}

Open a name netconf session, but don't send `hello`.

As *open/2* but does not send a `hello` message.

open(Options) -> Result

Types:

Options = options()
Result = {ok, handle()} | {error, error_reason()}

Open a netconf session and exchange `hello` messages.

If the server options are specified in a configuration file, or if a named client is needed for logging purposes (see *Logging*) use *open/2* instead.

The opaque `handler()` reference which is returned from this function is required as client identifier when calling any other function in this module.

The `timeout` option (milli seconds) is used when setting up the ssh connection and when waiting for the hello message from the server. It is not used for any other purposes during the lifetime of the connection.

open(KeyOrName, ExtraOptions) -> Result

Types:

KeyOrName = key_or_name() (see module ct_gen_conn)

```
ExtraOptions = options()
Result = {ok, handle()} | {error, error_reason()}
```

Open a named netconf session and exchange `hello` messages.

If `KeyOrName` is a configured `server_id()` or a `target_name()` associated with such an ID, then the options for this server will be fetched from the configuration file.

The `ExtraOptions` argument will be added to the options found in the configuration file. If the same options are given, the values from the configuration file will overwrite `ExtraOptions`.

If the server is not specified in a configuration file, use *open/1* instead.

The opaque `handle()` reference which is 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()`, i.e. if the server is named via a call to `ct:require/2` or a `require` statement in the test suite, then this name may be used instead of the `handle()`.

The `timeout` option (milli seconds) is used when setting up the ssh connection and when waiting for the hello message from the server. It is not used for any other purposes during the lifetime of the connection.

See also: ct:require/2.

`send(Client, SimpleXml) -> Result`

Equivalent to *send(Client, SimpleXml, infinity)*.

`send(Client, SimpleXml, Timeout) -> Result`

Types:

```
Client = client()
SimpleXml = simple_xml()
Timeout = timeout()
Result = simple_xml() | {error, error_reason()}
```

Send an XML document to the server.

The given XML document is sent as is to the server. This function can be used for sending XML documents that can not be expressed by other interface functions in this module.

`send_rpc(Client, SimpleXml) -> Result`

Equivalent to *send_rpc(Client, SimpleXml, infinity)*.

`send_rpc(Client, SimpleXml, Timeout) -> Result`

Types:

```
Client = client()
SimpleXml = simple_xml()
Timeout = timeout()
Result = [simple_xml()] | {error, error_reason()}
```

Send a Netconf `rpc` request to the server.

The given XML document is wrapped in a valid Netconf `rpc` request and sent to the server. The `message-id` and namespace attributes are added to the `rpc` element.

This function can be used for sending `rpc` requests that can not be expressed by other interface functions in this module.

unlock(Client, Target) -> Result

Equivalent to *unlock(Client, Target, infinity)*.

unlock(Client, Target, Timeout) -> Result

Types:

```
Client = client()
Target = netconf_db()
Timeout = timeout()
Result = ok | {error, error_reason()}
```

Unlock configuration target.

If the client earlier has aquired a lock, via *lock/2* or *lock/3*, this operation release the associated lock. To be able 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 the application the function will make the test case calling this function 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 `app_node/2` only the `FailOnBadRPC` argument will determine if the search for a candidate node should stop or not 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 `app_node/2` only the `FailOnBadRPC` argument will determine if the search for a candidate node should stop or not if `badrpc` is received at some point. The cookie on the client node will be set to `Cookie` for this rpc operation (use 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 whatever Function returns or {badrpc, Reason} if the remote procedure call fails. If Node is {Fun, FunArgs} applying Fun to FunArgs should 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 whatever Function returns or {badrpc, Reason} if the remote procedure call fails. If Node is {Fun, FunArgs} applying Fun to FunArgs should return a node name. The cookie on the client node will be set to `Cookie` for this rpc operation (use 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 which makes the call is not suspended until the evaluation is compleated as in the case of call/[3,4]. If Node is {Fun, FunArgs} applying Fun to FunArgs should 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 which makes the call is not suspended until the evaluation is compleated as in the case of call/[3,4]. If Node is {Fun, FunArgs} applying Fun to FunArgs should return a node name. The cookie on the client node will be set to `Cookie` for this rpc operation (use to match the server node cookie).

ct_snmp

Erlang module

Common Test user interface module for the OTP snmp application

The purpose of this module is to make snmp configuration easier 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 may be passed to `ct_snmp` by means of Common Test configuration files. For more specialized configuration parameters, it is possible to place a "simple snmp configuration file" 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 doesn't have to handle as many input parameters as it would if it had to interface the OTP snmp manager directly.

The following snmp manager and agent parameters are configurable:

```
{snmp,
       %%% Manager config
       [{start_manager, boolean()}    % Optional - default is true
       {users, [{user_name(), [call_back_module(), user_data()]}]}, %% Optional
       {usm_users, [{usm_user_name(), [usm_config()]}]},%% Optional - snmp v3 only
       % managed_agents is optional
       {managed_agents,[{agent_name(), [user_name(), agent_ip(), agent_port(), [agent_config()]]}]},
       {max_msg_size, integer()},     % Optional - default is 484
       {mgr_port, integer()},         % Optional - default is 5000
       {engine _id, string()},        % Optional - default is "mgrEngine"

       %%% Agent config
       {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.
       %% Note all values in agent.conf can be altered by the parametes
       %% above. All these configuration files have default values set
       %% up 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 dir that can produce a list
       %% of valid configuration values if you apply 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
      ]}.
```

The MgrAgentConfName parameter in the functions should 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).
```

Note that Usm users are needed for snmp v3 configuration and are not to be confused with users.

Snmp traps, inform and report messages are handled by the user callback module. For more information about this see the snmp application.

Note: It is recommended to use the .hrl-files created by the Erlang/OTP mib-compiler to define the oids. Example for the getting the erlang node name from the erlNodeTable in the OTP-MIB:

```
Oid = ?erlNodeEntry ++ [?erlNodeName, 1]
```

It is also possible to set values for snmp application configuration parameters, such as config, server, net_if, etc (see the "Configuring the application" chapter in the OTP snmp User's Guide for a list of valid parameters and types). 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 needs to be allocated in the suite using require (see example above), and this name passed as the SnmpAppConfName argument to start/3. ct_snmp specifies default values for some snmp application configuration parameters (such as {verbosity,trace} for the config parameter). This set of defaults will be merged with the parameters specified by the user, and 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()]
```

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()

Load the mibs into the agent 'snmp_master_agent'.

register_agents(MgrAgentConfName, ManagedAgents) -> ok | {error, Reason}

Types:

MgrAgentConfName = atom()
ManagedAgents = [agent()]
Reason = term()

Explicitly instruct the manager to handle this agent. Corresponds to making an entry in agents.conf

This function will try to register the given managed agents, without checking if any of them already exist. In order to change an already registered managed agent, the agent must first be unregistered.

register_users(MgrAgentConfName, Users) -> ok | {error, Reason}

Types:

MgrAgentConfName = atom()
Users = [user()]
Reason = term()

Register the manager entity (=user) responsible for specific agent(s). Corresponds to making an entry in users.conf.

This function will try to register the given users, without checking if any of them already exist. In order to change an already 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 instruct the manager to handle this USM user. Corresponds to making an entry in usm.conf

This function will try to register the given users, without checking if any of them already exist. In order to change an already 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 prior to the set and the new value. This is intended to facilitate the clean up in the end_per_testcase function i.e. the undoing of the set requests and its 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 *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` will be performed. When using snmp v3 also so called usm users will be registered. Note that users, usm_users and managed agents may also be registered at a later time using ct_snmp:register_users/2, ct_snmp:register_agents/2, and ct_snmp:register_usm_users/2. The agent started will be called `snmp_master_agent`. Use ct_snmp:load_mibs/1 to load mibs into the agent. With `SnmpAppConfName` it's possible to configure the snmp application with parameters such as `config`, `mibs`, `net_if`, etc. The values will be 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 removes all files created.

`unload_mibs(Mibs) -> ok | {error, Reason}`

Types:

```
Mibs = [MibName]
MibName = string()
Reason = term()
```

Unload the mibs from the agent 'snmp_master_agent'.

`unregister_agents(MgrAgentConfName) -> ok`

Types:

```
MgrAgentConfName = atom()
Reason = term()
```

Unregister all managed agents.

`unregister_agents(MgrAgentConfName, ManagedAgents) -> ok`

Types:

```
MgrAgentConfName = atom()
ManagedAgents = [agent_name()]
Reason = term()
```

Unregister the given managed agents.

`unregister_users(MgrAgentConfName) -> ok`

Types:

```
MgrAgentConfName = atom()
Reason = term()
```

Unregister all users.

unregister_users(MgrAgentConfName, Users) -> ok

Types:

MgrAgentConfName = atom()
Users = [user_name()]
Reason = term()

Unregister the given users.

unregister_usm_users(MgrAgentConfName) -> ok

Types:

MgrAgentConfName = atom()
Reason = term()

Unregister all usm users.

unregister_usm_users(MgrAgentConfName, UsmUsers) -> ok

Types:

MgrAgentConfName = atom()
UsmUsers = [usm_user_name()]
Reason = term()

Unregister the given 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. See the `unix_telnet` manual page for information about how to use `ct_telnet`, and configure connections, specifically for unix hosts.

The following default values are defined in `ct_telnet`:

```
Connection timeout = 10 sec (time to wait for connection)
Command timeout = 10 sec (time to wait for a command to return)
Max no of reconnection attempts = 3
Reconnection interval = 5 sek (time to wait in between reconnection attempts)
Keep alive = true (will send NOP to the server every 10 sec if connection is idle)
Polling limit = 0 (max number of times to poll to get a remaining string terminated)
Polling interval = 1 sec (sleep time between polls)
```

These parameters can be altered 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}]}.
```

`Millisec = integer(), N = integer()`

Enter the `telnet_settings` term in a configuration file included in the test and ct_telnet will retrieve the information automatically. Note that `keep_alive` may be specified per connection if required. See `unix_telnet` for details.

Logging

The default logging behaviour of `ct_telnet` is to print information to the test case HTML log about performed operations and commands and their corresponding results. What won't be printed to the HTML log are text strings sent from the telnet server that are not explicitly received by means of a `ct_telnet` function such as `expect/3`. `ct_telnet` may however be configured to use a special purpose event handler, implemented in `ct_conn_log_h`, for logging *all* telnet traffic. To use this handler, you need to install a Common Test hook named `cth_conn_log`. Example (using the test suite info 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, i.e. `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 arbitrary data sent from the server. The link to this text file can be found on the top of the test case HTML log.

By default, data for all telnet connections is logged in one common file (named `default`), which might get messy e.g. if multiple telnet sessions are running in parallel. It is therefore possible to create a separate log file for each connection. To configure this, use the hook option `hosts` and list the names of the servers/connections that will be used in the suite. Note that the connections must be named for this to work (see the `open` function below).

The hook option named `log_type` may be used to change the `cth_conn_log` behaviour. The default value of this option is `raw`, which results in the behaviour 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 above can also be specified in a configuration file with the configuration variable `ct_conn_log`. Example:

```
{ct_conn_log, [{ct_telnet,[{log_type,raw},
                           {hosts,[key_or_name()]}]}]}
```

Note that hook options specified in a configuration file will overwrite any hardcoded hook options in the test suite!

Logging example

The following `ct_hooks` statement will cause printing of telnet traffic to separate logs for the connections named `server1` and `server2`. Traffic for any other connections will be logged in the default telnet log.

```
suite() ->
    [{ct_hooks,
      [{cth_conn_log, [{ct_telnet,[{hosts,[server1,server2]}]}]}]}].
```

As previously explained, the above specification could also be provided by means of an entry like this in a configuration file:

```
{ct_conn_log, [{ct_telnet,[{hosts,[server1,server2]}]}]}.
```

in which case the `ct_hooks` statement in the test suite may simply look like this:

```
suite() ->
    [{ct_hooks, [{cth_conn_log, []}]}].
```

DATA TYPES

```
connection() = handle() | {target_name() (see module ct), connection_type()}
| target_name() (see module ct)
connection_type() = telnet | ts1 | ts2
handle() = handle() (see module ct_gen_conn)
```

Handle for a specific telnet connection.

```
prompt_regexp() = string()
```

A regular expression which matches all possible prompts for a specific type of target. The regexp must not have any groups i.e. when matching, re:run/3 shall return a list with one single element.

Exports

`close(Connection) -> ok | {error, Reason}`

Types:

```
Connection = connection() (see module ct_telnet)
Reason = term()
```

Close the telnet connection and stop the process managing it.

A connection may be associated with a target name and/or a handle. If `Connection` has no associated target name, it may only be closed with the handle value (see the `open/4` function).

`cmd(Connection, Cmd) -> {ok, Data} | {error, Reason}`

Equivalent to *cmd(Connection, Cmd, [])*.

`cmd(Connection, Cmd, Opts) -> {ok, Data} | {error, Reason}`

Types:

```
Connection = connection() (see module ct_telnet)
Cmd = string()
Opts = [Opt]
Opt = {timeout, timeout()} | {newline, boolean()}
Data = [string()]
Reason = term()
```

Send a command via telnet and wait for prompt.

This function will by default add a newline to the end of the given command. If this is not desired, the option `{newline,false}` can be used. This is necessary, for example, when sending telnet command sequences (prefixed with the Interprete As Command, IAC, character).

The option `timeout` specifies how long the client shall wait for prompt. If the time expires, the function returns `{error,timeout}`. See the module description for information about the default value for the command timeout.

`cmdf(Connection, CmdFormat, Args) -> {ok, Data} | {error, Reason}`

Equivalent to *cmdf(Connection, CmdFormat, Args, [])*.

`cmdf(Connection, CmdFormat, Args, Opts) -> {ok, Data} | {error, Reason}`

Types:

```
Connection = connection() (see module ct_telnet)
CmdFormat = string()
Args = list()
Opts = [Opt]
Opt = {timeout, timeout()} | {newline, boolean()}
Data = [string()]
Reason = term()
```

Send a telnet command and wait for prompt (uses a format string and list of arguments to build the command).

See *cmd/3* further description.

`expect(Connection, Patterns) -> term()`

Equivalent to *expect(Connections, Patterns, [])*.

expect(Connection, Patterns, Opts) -> {ok, Match} | {ok, MatchList, HaltReason} | {error, Reason}

Types:

```
Connection = connection() (see module ct_telnet)
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}
```

Get data from telnet and wait for the expected pattern.

`Pattern` can be a POSIX regular expression. The function returns as soon as a pattern has been successfully matched (at least one, in the case of multiple patterns).

`RxMatch` is a list of matched strings. It looks like this: `[FullMatch, SubMatch1, SubMatch2, ...]` where `FullMatch` is the string matched by the whole regular expression and `SubMatchN` is the string that matched subexpression no `N`. Subexpressions are denoted with '(' ')' in the regular expression

If a `Tag` is given, the returned `Match` will also include the matched `Tag`. Else, only `RxMatch` is returned.

The `idle_timeout` option indicates that the function shall return if the telnet client is idle (i.e. if no data is received) for more than `IdleTimeout` milliseconds. Default timeout is 10 seconds.

The `total_timeout` option sets a time limit for the complete expect operation. After `TotalTimeout` milliseconds, `{error,timeout}` is returned. The default value is `infinity` (i.e. no time limit).

The function will return when a prompt is received, even if no pattern has yet been matched. In this event, `{error, {prompt,Prompt}}` is returned. However, this behaviour may be modified with the `ignore_prompt` or `no_prompt_check` option, which tells `expect` to return only when a match is found or after a timeout.

If the `ignore_prompt` option is used, `ct_telnet` will ignore any prompt found. This option is useful if data sent by the server could include a pattern that would match the prompt regexp (as returned by `TargedMod:get_prompt_regexp/0`), but which should not cause the function to return.

If the `no_prompt_check` option is used, `ct_telnet` will not search for a prompt at all. This is useful if, for instance, the `Pattern` itself matches the prompt.

The `wait_for_prompt` option forces `ct_telnet` to wait until the prompt string has been received before returning (even if a pattern has already been matched). This is equal to calling: `expect(Conn, Patterns++[{prompt,Prompt}], [sequence|Opts])`. Note that `idle_timeout` and `total_timeout` may abort the operation of waiting for prompt.

The `repeat` option indicates that the pattern(s) shall be matched multiple times. If `N` is given, the pattern(s) will be matched `N` times, and the function will return with `HaltReason = done`.

The `sequence` option indicates that all patterns shall be matched in a sequence. A match will not be concluded untill all patterns are matched.

Both `repeat` and `sequence` can be interrupted by one or more `HaltPatterns`. When `sequence` or `repeat` is used, there will always be a `MatchList` returned, i.e. a list of `Match` instead of only one `Match`. There will also be a `HaltReason` returned.

Examples:
`expect(Connection,[{abc,"ABC"},{xyz,"XYZ"}], [sequence,{halt,[{nnn,"NNN"}]}]).`
will try to match "ABC" first and then "XYZ", but if "NNN" appears the function will return `{error,{nnn,["NNN"]}}`. If both "ABC" and "XYZ" are matched, the function will return `{ok,[AbcMatch,XyzMatch]}`.

`expect(Connection,[{abc,"ABC"},{xyz,"XYZ"}], [{repeat,2},{halt,[{nnn,"NNN"}]}]).`
will try to match "ABC" or "XYZ" twice. If "NNN" appears the function will return with `HaltReason = {nnn,["NNN"]}`.

The `repeat` and `sequence` options can be combined in order to match a sequence multiple times.

`format_data(How, X2) -> term()`

`get_data(Connection) -> {ok, Data} | {error, Reason}`

Types:

`Connection = connection() (see module ct_telnet)`
`Data = [string()]`
`Reason = term()`

Get all data that has been received by the telnet client since the last command was sent. Note that only newline terminated strings are returned. If the last string received has not yet been terminated, the connection may be polled automatically until the string is complete. The polling feature is controlled by the `poll_limit` and `poll_interval` config values and is by default disabled (meaning the function will immediately return all complete strings received and save a remaining non-terminated string for a later `get_data` call).

`open(Name) -> {ok, Handle} | {error, Reason}`

Equivalent to *open(Name, telnet)*.

`open(Name, ConnType) -> {ok, Handle} | {error, Reason}`

Types:

`Name = target_name()`
`ConnType = connection_type() (see module ct_telnet)`
`Handle = handle() (see module ct_telnet)`
`Reason = term()`

Open a telnet connection to the specified target host.

`open(KeyOrName, ConnType, TargetMod) -> {ok, Handle} | {error, Reason}`

Equivalent to *open(KeyOrName, ConnType, TargetMod, [])*.

open(KeyOrName, ConnType, TargetMod, Extra) -> {ok, Handle} | {error, Reason}

Types:

KeyOrName = Key | Name
Key = atom()
Name = target_name() (see module ct)
ConnType = connection_type()
TargetMod = atom()
Extra = term()
Handle = handle()
Reason = term()

Open a telnet connection to the specified target host.

The target data must exist in a configuration file. The connection may be associated with either `Name` and/or the returned `Handle`. To allocate a name for the target, use `ct:require/2` in a test case, or use a `require` statement in the suite info function (`suite/0`), or in a test case info function. If you want the connection to be associated with `Handle` only (in case you need to open multiple connections to a host for example), simply use `Key`, the configuration variable name, to specify the target. Note that a connection that has no associated target name can only be closed with the handle value.

`TargetMod` is a module which exports the functions `connect(Ip,Port,KeepAlive,Extra)` and `get_prompt_regexp()` for the given `TargetType` (e.g. `unix_telnet`).

See also: ct:require/2.

send(Connection, Cmd) -> ok | {error, Reason}

Equivalent to *send(Connection, Cmd, [])*.

send(Connection, Cmd, Opts) -> ok | {error, Reason}

Types:

Connection = connection() (see module ct_telnet)
Cmd = string()
Opts = [Opt]
Opt = {newline, boolean()}
Reason = term()

Send a telnet command and return immediately.

This function will by default add a newline to the end of the given command. If this is not desired, the option `{newline,false}` can be used. This is necessary, for example, when sending telnet command sequences (prefixed with the Interprete As Command, IAC, character).

The resulting output from the command can be read with `get_data/1` or `expect/2/3`.

sendf(Connection, CmdFormat, Args) -> ok | {error, Reason}

Equivalent to *sendf(Connection, CmdFormat, Args, [])*.

sendf(Connection, CmdFormat, Args, Opts) -> ok | {error, Reason}

Types:

Connection = connection() (see module ct_telnet)

```
CmdFormat = string()
Args = list()
Opts = [Opt]
Opt = {newline, boolean()}
Reason = term()
```

Send a telnet command and return immediately (uses a format string and a list of arguments to build the command).

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 config file:

```
{unix,[{telnet,HostNameOrIpAddress},
       {port,PortNum},                % optional
       {username,UserName},
       {password,Password},
       {keep_alive,Bool}]}.           % optional
```

To communicate via telnet to the host specified by `HostNameOrIpAddress`, use the interface functions in `ct_telnet`, e.g. `open(Name), cmd(Name,Cmd), ...`.

`Name` is the name you allocated to the unix host in your `require` statement. E.g.

```
suite() -> [{require,Name,{unix,[telnet]}}].
```

or

```
ct:require(Name,{unix,[telnet]}).
```

The "keep alive" activity (i.e. that Common Test sends NOP to the server every 10 seconds if the connection is idle) may be enabled or disabled for one particular connection as described here. It may be disabled for all connections using `telnet_settings` (see `ct_telnet`).

Note that the `{port,PortNum}` tuple is optional and if omitted, default telnet port 23 will be used. Also the `keep_alive` tuple is optional, and the value defauls to true (enabled).

Exports

`connect(ConnName, Ip, Port, Timeout, KeepAlive, Extra) -> {ok, Handle} | {error, Reason}`

Types:

```
ConnName = target_name() (see module ct)
Ip = string() | {integer(), integer(), integer(), integer()}
Port = integer()
Timeout = integer()
KeepAlive = bool()
Extra = target_name() (see module ct) | {Username, Password}
Username = string()
Password = string()
Handle = handle() (see module ct_telnet)
Reason = term()
```

Callback for ct_telnet.erl.

Setup telnet connection to a unix host.

`get_prompt_regexp() -> PromptRegexp`

Types:

`PromptRegexp = prompt_regexp() (see module ct_telnet)`

Callback for ct_telnet.erl.

Return a suitable regexp string that will match common prompts for users on unix hosts.

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 which are used by the Common Test Master to start and stop "slave" nodes. It is the default callback module for the `{init, node_start}` term of 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: 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's list-valued.

See also: 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` will be used to log in onto the remote host `Host`. Username, if omitted, defaults to the current user name, and password is empty by default.

A list of functions specified in the `Startup` option will be executed after startup of the node. Note that all used modules should be present in the code path on the `Host`.

The timeouts are applied as follows:

- `BootTimeout` - time to start the Erlang node, in seconds. Defaults to 3 seconds. If node does not become pingable within this time, the result `{error, boot_timeout, NodeName}` is returned;
- `InitTimeout` - time to wait for the node until it calls the internal callback function informing master about successfull startup. Defaults to one second. In case of timed out message the result `{error, init_timeout, NodeName}` is returned;
- `StartupTimeout` - time to wait intil the node finishes to run the `StartupFunctions`. Defaults to one second. If this timeout occurs, the result `{error, startup_timeout, NodeName}` is returned.

Option `monitor_master` specifies, if the slave node should be stopped in case of master node stop. Defaults to false.

Option `kill_if_fail` specifies, if the slave node should be killed in case of a timeout during initialization or startup. Defaults to true. Note that node also may be still alive it the boot timeout occurred, but it will not be killed in this case.

Option `erlang_flags` specifies, which flags will be added to the parameters of the `erl` executable.

Option `env` specifies a list of environment variables that will extended the environment.

Special return values are:

- `{error, already_started, NodeName}` - if the node with the given name is already started on a given host;
- `{error, started_not_connected, NodeName}` - if node is started, but not connected to the master node.
- `{error, not_alive, NodeName}` - if node on which the `ct_slave:start/3` is called, is not alive. Note that `NodeName` is the name of 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 localhost.

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* (henceforth called CTH) framework allows extensions of the default behaviour of Common Test by means of callbacks before and after all test suite calls. It is meant for advanced users of Common Test which want to abstract out behaviour which is common to multiple test suites.

In brief, Common Test Hooks allows you to:

- Manipulate the runtime config before each suite configuration call
- Manipulate the return of all suite configuration calls and in extension the result of the test themselves.

The following sections describe the mandatory and optional CTH functions Common Test will call during test execution. For more details see *Common Test Hooks* in the User's Guide.

For information about how to add a CTH to your suite see *Installing a CTH* in the User's Guide.

> Note:
>
> See the *Example CTH* in the User's Guide for a minimal example of a CTH.

CALLBACK FUNCTIONS

The following functions define the callback interface for a Common Test Hook.

Exports

`Module:init(Id, Opts) -> {ok, State} | {ok, State, Priority}`

Types:

```
Id = reference() | term()
Opts = term()
State = term()
Priority = integer()
```

MANDATORY

Always called before any other callback function. Use this to initiate any common state. It should return a state for this CTH.

`Id` is the return value of *id/1*, or a `reference` (created using *make_ref/0*) if *id/1* is not implemented.

`Priority` is the relative priority of this hook. Hooks with a lower priority will be executed first. If no priority is given, it will be set to 0.

For details about when init is called see *scope* in the User's Guide.

`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 which needs to 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 will be 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 config 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* will never be called, instead the initiation is considered to be skipped/failed respectively. If a `NewConfig` list is returned, *init_per_suite* will be called with that `NewConfig` list. See *Pre Hooks* in the User's Guide for more details.

Note that this function is only called if the CTH has been added before init_per_suite is run, see *CTH Scoping* in the User's Guide for details.

`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 make sure that all the correct dependencies have been started correctly.

`Return` is what *init_per_suite* returned, i.e. {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* . It is here possible to recover from a failure in *init_per_suite* by returning the `ConfigList` with the `tc_status` element removed. See *Post Hooks* in the User's Guide for more details.

`CTHState` is the current internal state of the CTH.

Note that this function is only called if the CTH has been added before or in init_per_suite, see *CTH Scoping* in the User's Guide for details.

`Module:pre_init_per_group(GroupName, InitData, CTHState) -> Result`

Types:

```
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 the *init_per_group* instead.

`Module:post_init_per_group(GroupName, Config, Return, CTHState) -> Result`

Types:

```
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 the *init_per_group* instead.

`Module:pre_init_per_testcase(TestcaseName, InitData, CTHState) -> Result`

Types:

```
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 the *init_per_testcase* function instead.

Note that it is not possible to add CTH's here right now, that feature might be added later, but it would right now break backwards compatibility.

`Module:post_end_per_testcase(TestcaseName, Config, Return, CTHState) -> Result`

Types:

```
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_init_per_suite*, but for the *end_per_testcase* function instead.

`Module:pre_end_per_group(GroupName, EndData, CTHState) -> Result`

Types:

```
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 the *end_per_group* function instead.

`Module:post_end_per_group(GroupName, Config, Return, CTHState) -> Result`

Types:

```
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 the *end_per_group* function instead.

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 the *end_per_suite* function 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 the *end_per_suite* function instead.

Module:on_tc_fail(TestName, Reason, CTHState) -> NewCTHState

Types:

```
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 config function) fails. It is called after the post function has been called for the failed test case. I.e. if init_per_suite fails, this function is called after *post_init_per_suite*, and if a test case fails, it 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 simply the function name.

The data which comes with the Reason follows the same format as the *FailReason* in the *tc_done* event. See *Event Handling* in the User's Guide for details.

`Module:on_tc_skip(TestName, Reason, CTHState) -> NewCTHState`

Types:

```
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 config function) is skipped. It is called after the post function has been called for the skipped test case. I.e. if init_per_group is skipped, this function is called after *post_init_per_group*, and if a test case is skipped, it 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 simply the function name.

The data which comes with the Reason follows the same format as *tc_auto_skip* and *tc_user_skip* events. See *Event Handling* in the User's Guide for details.

`Module:terminate(CTHState)`

Types:

```
CTHState = term()
```

OPTIONAL

This function is called at the end of a CTH's *scope*.

`Module:id(Opts) -> Id`

Types:

```
Opts = term()
Id = term()
```

OPTIONAL

The `Id` is used to uniquely identify a CTH instance, if two CTH's return the same `Id` the second CTH is ignored and subsequent calls to the CTH will only be made to the first instance. For more information see *Installing a CTH* in the User's Guide.

This function should NOT have any side effects as it might be called multiple times by Common Test.

If not implemented the CTH will act 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 towards running Property Based testing 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 testsuite calling a property testing tool with special property test suites as defined by that tool. In this manual we assume the usual Erlang Application directory structure. The tests are collected in the application's `test` directory. The test directory has a sub-directory called `property_test` where everything needed for the property tests are collected.

A typical ct 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 which may be changed or removed anytime without any warning.

Exports

`init_per_suite(Config) -> Config | {skip, Reason}`

Initializes Config for property testing.

The function investigates if support is available for either Quickcheck, PropEr, or Triq. The options `{property_dir,AbsPath}` and `{property_test_tool,Tool}` is set in the Config returned.

The function is intended to be called in the init_per_suite in the test suite.

The property tests are assumed to be in the subdirectory `property_test`.

`quickcheck(Property, Config) -> true | {fail, Reason}`

Call quickcheck and return the result in a form suitable for common_test.

The function is intended to be called in the test cases in the test suite.