Common Test

Common Test 1.4.7

February 22 2010

February 22 2010

1 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 can support implementation and automated execution of test cases towards different types of target systems. The framework is based on the OTP Test Server. Test cases can be run individually or in batches. Common Test also features a distributed testing mode with central control and logging. This feature makes it possible to test multiple systems independently in one common session. This can be very 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 the CT GUI or from an OS- or Erlang shell prompt. *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.

The main idea is that CT based test programs connect to the target system(s) via standard O&M interfaces. CT provides implementations and wrappers of some of these O&M interfaces and will be extended with more interfaces later. 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 generators etc.

Common Test is also a very useful tool for white-box testing Erlang code since the test programs can call Erlang API functions directly. For black-box testing Erlang software, Erlang RPC as well as standard O&M interfaces can 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!

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 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)

init_per_group(GroupName, Config)
: Configuration function for a group, executed before the first test case. (Mandatory if groups are defined)

end_per_group(GroupName, Config)
: Configuration function for a group, executed after the last test case. (Mandatory if groups are defined)

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 Installation

1.2.1 General information

The two main interfaces for running tests with Common Test are an executable Bourne shell script named `run_test` and an erlang module named `ct`. The shell script will work on Unix/Linux (and Linux-like environments such as Cygwin on Windows) and the `ct` interface functions can be called from the Erlang shell (or from any Erlang function) on any supported platform.

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

1.2.2 Unix/Linux

Go to the `common_test-<vsn>` directory, located among the other OTP applications (under the OTP lib directory). Here you 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.

For more information on the `run_test` script and the `ct` module, please see the reference manual.

1.2.3 Windows

On Windows it is very convenient to use Cygwin (`www.cygwin.com`) for running Common Test and Erlang, since it enables you to use the `run_test` script for starting Common Test. If you are a Cygwin user, simply follow the instructions above for generating the `run_test` script.

If you do not use Cygwin, you have to rely on the API functions in the `ct` module (instead of `run_test`) for running Common Test as described initially in this chapter.

If you, for any reason, have chosen to store Common Test and Test Server in a different location than the default OTP lib directory, make sure the `ebin` directories of these applications are included in the Erlang code server path (so the application modules can be loaded).

1.3 Writing Test Suites

1.3.1 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_<something>` that provide various support, mainly simplified use of communication protocols such as rpc, snmp, ftp, telnet, etc.

1.3.2 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).

The `ct.hrl` header file must be included in all test suite files.

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

1.3.3 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`.

1.3.4 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).

If `init_per_testcase` crashes, the test case itself is skipped automatically (so called *auto skipped*). If `init_per_testcase` returns a `skip` tuple, also then will the test case be skipped (so called *user skipped*). In either event, the `end_per_testcase` is never 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.

1.3.5 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)`.

1.3.6 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.

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, [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"}]}
```

```
        ].
```

1.3.7 Test suite info function

The `suite/0` function can be used in a test suite module to set the default values for the `timetrap` and `require` tags. If a test case info function also specifies any of these tags, the default value is overruled. See above for more information.

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"}
          ].
```

1.3.8 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}].
```

Properties may be combined so that e.g. if `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, when 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 function is mandatory if one or more test case groups are defined). 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`.

> **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`).

1.3.9 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.

1.3.10 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.

1.3.11 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).

1.3.12 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.

The `priv_dir` is the test suite's private directory. This directory should be used when a test case needs to write to files. The name of the private directory is generated by the test server, which also creates the directory.

> 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.

1.3.13 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.

The default time limit for a test case is 30 minutes, unless a `timetrap` is specified either by the test case info function or the `suite/0` function.

1.3.14 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 the Clearcase (file version control system) paths and files:

  The test suites are stored in Clearcase but are not (necessarily) run within this environment. The directory structure may vary from test run to test run.
- Depending on execution order:

  During development of test suites, no assumption should be made (preferrably) 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.

1.4 Test Structure

1.4.1 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.

1.4.2 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.

1.4.3 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.

run_test
: The name of an executable Bourne shell script that may be used on Linux/Unix 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.

1.5 Examples and Templates

1.5.1 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.
```

1.5.2 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.
```

1.6 Running Test Suites

1.6.1 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.

1.6.2 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.

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 `run_test`, 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 `run_test`, 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. Common Test will only verify that the specified test suites exist before starting the tests.

1.6.3 Running tests from the UNIX command line

The script `run_test` can be used for running tests from the Unix/Linux command line, e.g.

- `run_test -config <configfilenames> -dir <dirs>`
- `run_test -config <configfilenames> -suite <suiteswithfullpath>`
- `run_test -config <configfilenames> -suite <suitewithfullpath> -group <groupnames> -case <casenames>`

Examples:

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

Other flags that may be used with `run_test`:

- `-logdir <dir>`, specifies where the HTML log files are to be written.
- `-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*).
- `-event_handler <event_handlers>`, to install *event handlers*.
- `-include`, specifies include directories (see above).
- `-no_auto_compile`, disables the automatic test suite compilation feature (see above).
- `-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`, on timeout, the test run will be aborted when current test job is finished (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.

> 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 `run_test`. 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:
>
> ```
> $ run_test -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!

For details on how to generate the `run_test` script, see the *Installation* chapter.

1.6.4 Running tests from the Web based GUI

The web based GUI, VTS, is started with the `run_test` script. 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.

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

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

Note that `run_test -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:

```
run_test -vts -browser <browser_start_cmd>
```

Example:

```
$ run_test -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, netscape will be the default browser on Unix platforms and Internet Explorer on Windows. If Common Test fails to start a browser automatically, start your favourite browser manually instead and type in the URL that Common Test displays in the shell.

1.6.5 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 `run_test` script described above, only the flags are instead given as options in a list of key-value tuples. E.g. a test specified with `run_test` like:

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

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

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

For detailed documentation, please see the `ct` manual page.

1.6.6 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 `run_test` script, you may start the Erlang shell and Common Test in the same go by using the `-shell` and, optionally, the `-config` flag:

- `run_test -shell`
- `run_test -shell -config <configfilename>`

If no config file is given with the `run_test` 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 `run_test`. 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.

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

By means of `run_test -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_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`.

1.6.8 Using test specifications

The most expressive 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 may be declared in a text file or passed to the test server at runtime as a list (see `run_testspec/1` in the manual page for `ct`). There are two general types of terms: configuration terms and test specification terms.

With configuration terms it is possible to import configuration data (similar to `run_test -config`), specify HTML log directories (similar to `run_test -logdir`), give aliases to test nodes and test directories (to make a specification easier to read and maintain), enable code coverage analysis (see the *Code Coverage Analysis* chapter) and specify event_handler plugins (see the *Event Handling* chapter). There is also a term for specifying include directories that should be passed on to the compiler when automatic compilation is performed (similar to `run_test -include`, see above).

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 or one or more test cases. An arbitrary number of test terms may be declared in sequence. A test term can also specify one or more test suites or test cases to be skipped. Skipped suites and cases are not executed and show up in the HTML test log as SKIPPED.

> **Note:**
>
> It is not yet possible to specify test case groups in test specifications. This will be supported in a soon upcoming release.

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. Node parameters are only relevant in the latter (see the chapter about running Common Test in distributed mode for information). For details on the event_handler term, see the *Event Handling* chapter.

Config terms:

```
{node, NodeAlias, Node}.

{cover, CoverSpecFile}.
{cover, NodeRef, CoverSpecFile}.

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

{config, ConfigFiles}.
{config, NodeRefs, ConfigFiles}.

{alias, DirAlias, Dir}.

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

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

Test terms:

```
{suites, DirRef, Suites}.
{suites, NodeRefs, DirRef, Suites}.

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

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

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

Types:

```
NodeAlias     = atom()
Node          = node()
NodeRef       = NodeAlias | Node | master
NodeRefs      = all_nodes | [NodeRef] | NodeRef
CoverSpecFile = string()
IncludeDirs   = string() | [string()]
ConfigFiles   = string() | [string()]
DirAlias      = atom()
Dir           = string()
LogDir        = string()
EventHandlers = atom() | [atom()]
InitArgs      = [term()]
DirRef        = DirAlias | Dir
Suites        = atom() | [atom()] | all
Cases         = atom() | [atom()] | all
Comment       = string() | ""
```

Example:

```
    {logdir, "/home/test/logs"}.

    {config, "/home/test/t1/cfg/config.cfg"}.
    {config, "/home/test/t2/cfg/config.cfg"}.
    {config, "/home/test/t3/cfg/config.cfg"}.

    {alias, t1, "/home/test/t1"}.
    {alias, t2, "/home/test/t2"}.
    {alias, t3, "/home/test/t3"}.

    {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.
- Aliases are given for three test system directories. The suites in this example are stored in "/home/test/tX/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.

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 `run_test`. 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}`

1.6.9 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 logfile 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 file contain full details of every single test case, each one in a separate file. This way the files should be easy to compare with previous test runs, even if the set of test cases change.

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.

1.6.10 HTML Style Sheets

Common Test includes the *optional* feature to use HTML style sheets (CSS) for customizing user 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:

```
<style>
  div.ct_internal { background:lightgrey; color:black }
  div.default     { background:lightgreen; color:black }
  div.sys_config  { background:blue; color:white }
  div.sys_state   { background:yellow; color:black }
  div.error       { background:red; color:white }
</style>
```

To install the CSS file (Common Test inlines the definition in the HTML code), the name may be provided when executing `run_test`. Example:

```
  $ run_test -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).

If the `Category` argument is not specified, Common Test will use the CSS selector `div.default` for the printout. For this reason a user supplied style sheet must include this selector. Also the selector `div.ct_internal` must be

included. Hence a minimal user style sheet should look like this (which is also the default style sheet Common Test uses if no user CSS file is provided):

```
<style>
div.ct_internal { background:lightgrey; color:black }
div.default     { background:lightgreen; color:black }
</style>
```

1.6.11 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 `run_test` 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})`

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 test run immediately, since this might leave the system under test in an undefined, and possibly bad, state. Instead Common Test will finish the current test job, or the complete test run, before stopping. The latter is the default behaviour. The `force_stop` flag/option tells Common Test to stop as soon as the current test job is finished. Note that since Common Test always finishes off the current test job or test session, 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:

```
$ run_test -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:

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

        $ run_test -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 3:

```
        $ run_test -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.

1.6.12 Silent Connections

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

```
run_test -silent_connections [conn_types]
```

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

Example:

```
run_test ... -silent_connections telnet ftp
```

switches off logging for telnet and ftp connections.

```
run_test ... -silent_connections
```

switches off logging for all connection types.

Basic and important information such as opening and closing a connection, 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 may 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

(`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.

The `silent_connections` setting returned from a test case info function overrides, for the test case in question, any setting made with `suite/0` (which is the setting used for all cases in the suite). Example:

```
-module(my_SUITE).
...
suite() -> [..., {silent_connections,[telnet,ftp]}, ...].
...
my_testcase1() ->
[{silent_connections,[ftp]}].
my_testcase1(_) ->
...
my_testcase2(_) ->
...
```

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

The `-silent_connections` tag (or `silent_connections` tagged tuple in the call to `ct:run_test/1`) overrides any settings in the test suite.

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

1.7 Config Files

1.7.1 General

The Common Test framework uses configuration files to describe data related to a test and/or an SUT (System Under Test). The configuration data makes it possible to change properties without changing the test program itself. Configuration data can for example be:

- Addresses to the test plant or other instruments
- Filenames for files needed by the test
- Program names for programs that shall be run by the test
- Any other variable that is needed by the test

1.7.2 Syntax

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

```
    {Key,Value}.
```

where

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

1.7.3 Requiring and reading configuration data

In a test suite, one must *require* that a configuration variable exists before attempting to read the associated value in a test case.

`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,Required}` or `{require,Name,Required}`. The arguments `Name` and `Required` are the same as the arguments to `ct:require/[1,2]` which are described in the reference manual for *ct*. `Name` becomes an alias for the configuration variable `Required`, 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),
...
```

1.7.4 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.

1.7.5 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.

1.7.6 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).

1.7.7 Examples

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"}.
```

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 Code Coverage Analysis

1.8.1 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.

1.8.2 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.

Note: Currently, for Common Test to be able to print code coverage HTML files for the modules included in the analysis, the source code files of these modules must be located in the same directory as the corresponding `.beam` files. This is a limitation that will be removed later.

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. Note that even if you run separate tests in one test run, code coverage data will not be passed on from one test to another unless you specify an export file for Common Test to use for this purpose.

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 `run_test`. Example:

```
$ run_test -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*).

1.8.3 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}.
```

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).

1.8.4 Logging

To view the result of a code coverage test, follow the "Coverage log" link on the test suite results page. This 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.

1.9 Using Common Test for Large Scale Testing

1.9.1 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.

1.9.2 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.

1.9.3 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). 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:

```
{node, node1, ct_node@host_x}.
{node, node2, ct_node@host_y}.

{logdir, master, "/home/test/master_logs"}.
{logdir, "/home/test/logs"}.

{config, node1, "/home/test/t1/cfg/config.cfg"}.
{config, node2, "/home/test/t2/cfg/config.cfg"}.
{config, "/home/test/t3/cfg/config.cfg"}.

{alias, t1, "/home/test/t1"}.
{alias, t2, "/home/test/t2"}.
{alias, t3, "/home/test/t3"}.

{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 `run_test` script), will be performed. Tests without explicit node association will always be performed too of course!

> Note:
>
> It is recommended that absolute paths are used for log directories, config files and test directory aliases in the test specifications so that current working directory settings are not important.

1.10 Event Handling

1.10.1 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.

1.10.2 Usage

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

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

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).

It is not possible to specify start arguments to the event handlers when using the `run_test` script. You may however pass along start arguments if you use the `ct:run_test/1` function. 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_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 | {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()}`, 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,Func,Reason}}`

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

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

  `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()}`, 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.

- `#event{name = tc_user_skip, data = {Suite,TestCase,Comment}}`

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

  `TestCase = atom()`, name of the test case.

  `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.

- `#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.

1.11 Dependencies between Test Cases and Suites

1.11.1 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.
```

1.11.2 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.

1.11.3 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.

1.12 Some thoughts about testing

1.12.1 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.

1.12.2 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() -> TestCases | {skip,Reason}`

Types:

TestCases = [atom() | {group,GroupName}]

Reason = term()

GroupName = atom()

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,GroupName}` tuple, where `GroupName`, an atom, is the name of the group (defined with `groups/0`).

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

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}

Time = MilliSec | {seconds,integer()} | {minutes,integer()} | {hours,integer()}

MilliSec = integer()

Required = Key | {Key,SubKeys}

Key = atom()

SubKeys = SubKey | [SubKey]

SubKey = atom()

Name = atom()

UserData = term()

Conns = [atom()]

CSSFile = string()

OPTIONAL

This is the test suite info function. It is supposed to return a list of tagged tuples that specify various properties regarding 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 take (including `init_per_testcase/2` and `end_per_testcase/2`). If the timetrap time is exceeded, the test case fails with reason `timetrap_timeout`.

The `require` tag specifies configuration variables that are required by test cases 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`.

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 function is called as the first function in the suite. It typically contains initialization which is common for all test cases in the suite, and which shall only be done once. The `Config` parameter is the configuration 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:init_per_group(GroupName, Config) -> NewConfig | {skip,Reason}`

Types:

GroupName = atom()

Config = NewConfig = [{Key,Value}]

Key = atom()

Value = term()

Reason = term()

MANDATORY (only if one or more groups are defined)

This function is called before execution of a test case group. It typically contains initialization which is common for all test cases 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 which can be modified here. Whatever is returned from this function is given as `Config` to all test cases 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

MANDATORY (only if one or more groups are defined)

This function is called after the execution of a test case group is finished. It is meant to be used for cleaning up after `init_per_group/2`. 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 | {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` is the configuration which can be modified here. Whatever is returned from this function is given as `Config` to the test case. 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 = MilliSec | {seconds,integer()} | {minutes,integer()} | {hours,integer()}

MilliSec = integer()

Required = Key | {Key,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 regarding the execution of this particular test case.

The `timetrap` tag sets the maximum time the test case is allowed to take. 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.

The `require` tag specifies configuration variables that are required by the test case. 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 in the `suite/0` return list 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/[0,1]` (which also causes the test case process to crash).

Elements from the `Config` parameter can be read with the `?config` macro. The `config` macro is 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.

run_test

Command

The `run_test` script is automatically generated as Common Test is installed (please see the Installation chapter in the Common Test User's Guide for more information). The script accepts a number of different start flags. Some flags trigger `run_test` 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.

`run_test` also accepts Erlang emulator flags. These are used when `run_test` 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).

If `run_test` is called without parameters, it prints all valid start flags to stdout.

Run tests from command line

```
    run_test [-dir TestDir1 TestDir2 .. TestDirN] |
[-suite Suite1 Suite2 .. SuiteN
 [[-group Group1 Group2 .. GroupN] [-case Case1 Case2 .. CaseN]]]
[-step [config | keep_inactive]]
[-config ConfigFile1 ConfigFile2 .. ConfigFileN]
[-decrypt_key Key] | [-decrypt_file KeyFile]
[-logdir LogDir]
[-silent_connections [ConnType1 ConnType2 .. ConnTypeN]]
[-stylesheet CSSFile]
[-cover CoverCfgFile]
[-event_handler EvHandler1 EvHandler2 .. EvHandlerN]
[-include InclDir1 InclDir2 .. InclDirN]
[-no_auto_compile]
       [-repeat N [-force_stop]] |
       [-duration HHMMSS [-force_stop]] |
       [-until [YYMoMoDD]HHMMSS [-force_stop]]
[-basic_html]
```

Run tests using test specification

```
run_test -spec TestSpec1 TestSpec2 .. TestSpecN
[-config ConfigFile1 ConfigFile2 .. ConfigFileN]
[-decrypt_key Key] | [-decrypt_file KeyFile]
[-logdir LogDir]
[-allow_user_terms]
[-silent_connections [ConnType1 ConnType2 .. ConnTypeN]]
[-stylesheet CSSFile]
[-cover CoverCfgFile]
[-event_handler EvHandler1 EvHandler2 .. EvHandlerN]
[-include InclDir1 InclDir2 .. InclDirN]
[-no_auto_compile]
       [-repeat N [-force_stop]] |
       [-duration HHMMSS [-force_stop]] |
       [-until [YYMoMoDD]HHMMSS [-force_stop]]
[-basic_html]
```

Run tests in web based GUI

```
       run_test -vts [-browser Browser]
[-config ConfigFile1 ConfigFile2 .. ConfigFileN]
[-decrypt_key Key] | [-decrypt_file KeyFile]
       [-dir TestDir1 TestDir2 .. TestDirN] |
       [-suite Suite [[-group Group] [-case Case]]]
[-include InclDir1 InclDir2 .. InclDirN]
[-no_auto_compile]
[-basic_html]
```

Refresh the HTML index files

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

Run CT in interactive mode

```
run_test -shell
[-config ConfigFile1 ConfigFile2 ... ConfigFileN]
[-decrypt_key Key] | [-decrypt_file KeyFile]
```

Start an Erlang node with a given name

```
run_test -ctname NodeName
```

Start a Common Test Master node

```
run_test -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, no_testcase_running}`

Types:

Reason = term()

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.

`comment(Comment) -> void()`

Types:

Comment = term()

Print the given `Comment` in the comment field of the table on the test suite result page.

If called several times, only the last comment is printed. `comment/1` is also overwritten by the return value `{comment,Comment}` or by the function `fail/1` (which prints `Reason` as a comment).

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`.

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 = 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},
       {username,Username},
       {password,Password}]}.
```

```
get_config(unix,Default)     ->     [{telnet,IpAddr},     {username,Username},
{password,Password}]
get_config({unix,telnet},Default) -> IpAddr
get_config({unix,ftp},Default) -> Default
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:

```
require(myhost,unix) -> ok
get_config(myhost,Default)     ->     [{telnet,IpAddr},     {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 `{KeyOrName,Value}`, or (in case a subkey has been specified) `{{KeyOrName,SubKey},Value}`

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

`get_status() -> TestStatus | {error, Reason}`

Types:

TestStatus = [StatusElem]

StatusElem = {current, {Suite, TestCase}} | {successful, Successful} | {failed, Failed} | {skipped, Skipped} | {total, Total}

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, 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.

`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 `run_test` script.

`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, Format, [])*.

`log(X1, X2) -> ok`

Types:

X1 = Category | Format

X2 = Format | Args

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

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

Types:

Category = atom()

Format = string()

Args = list()

Printout from a testcase to the log.

This function is meant for printing stuff directly from a testcase (i.e. not from within the CT framework) in the test log.

Default `Category` is `default` and default `Args` is `[]`.

`pal(Format) -> ok`

Equivalent to *pal(default, Format, [])*.

`pal(X1, X2) -> ok`

Types:

X1 = Category | Format

X2 = Format | Args

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

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

Types:

Category = atom()

Format = string()

Args = list()

Print and log from a testcase.

This function is meant for printing stuff from a testcase both in the log and on the console.

Default `Category` is `default` and default `Args` is `[]`.

`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, Format, [])*.

`print(X1, X2) -> term()`

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

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

Types:

Category = atom()

Format = string()

Args = list()

Printout from a testcase to the console.

This function is meant for printing stuff from a testcase on the console.

Default `Category` is `default` and default `Args` is `[]`.

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

Types:

Required = Key | {Key, SubKeys}

Key = atom()

SubKeys = SubKey | [SubKey]

SubKey = atom()

Check if the required configuration is available.

Example: require the variable `myvar`:

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

In this case the config file must at least contain:

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

Example: require the variable `myvar` with subvariable `sub1`:

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

In this case the config file must at least contain:

```
{myvar,[{sub1,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, SubKeys}

Key = atom()

SubKeys = SubKey | [SubKey]

SubKey = atom()

Check if the required configuration is available, and give it a name.

If the requested data is available, the main 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 `Key`.

Example: Require one node with a telnet connection and an ftp connection. Name the node `a`:
`ok = ct:require(a,{node,[telnet,ftp]}).`
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:

```
{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 testcases in all suites in the given directories.

See also: run/3.

run(TestDir, Suite) -> Result

Run all testcases 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 testcase(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 = {config, CfgFiles} | {dir, TestDirs} | {suite, Suites} | {testcase, Cases} | {group, Groups} | {'spec', TestSpecs} | {allow_user_terms, Bool} | {logdir, LogDir} | {silent_connections, Conns} | {cover, CoverSpecFile} | {step, StepOpts} | {event_handler, EventHandlers} | {include, InclDirs} | {auto_compile, Bool} | {repeat, N} | {duration, DurTime} | {until, StopTime} | {force_stop, Bool} | {decrypt, DecryptKeyOrFile} | {refresh_logs, LogDir} | {basic_html, Bool}

CfgFiles = [string()] | string()

TestDirs = [string()] | string()

Suites = [string()] | string()

Cases = [atom()] | atom()

Groups = [atom()] | atom()

TestSpecs = [string()] | string()

LogDir = string()

Conns = all | [atom()]

CoverSpecFile = string()

StepOpts = [StepOpt] | []

StepOpt = config | keep_inactive

EventHandlers = EH | [EH]

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

InitArgs = [term()]

InclDirs = [string()] | string()

N = integer()

DurTime = string(HHMMSS)

StopTime = string(YYMoMoDDHHMMSS) | string(HHMMSS)

DecryptKeyOrFile = {key, DecryptKey} | {file, DecryptFile}

DecryptKey = string()

DecryptFile = string()

Result = [TestResult] | {error, Reason}

Run tests as specified by the combination of options in `Opts`. The options are the same as those used with the `run_test` script. 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 `run_test` script. Configuration files specified in `Opts` will be installed automatically at startup.

`run_testspec(TestSpec) -> Result`

Types:

TestSpec = [term()]

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

`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 unix command line with `run_test -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.

`testcases(TestDir, Suite) -> Testcases | {error, Reason}`

Types:

TestDir = string()

Suite = atom()

Testcases = list()

Reason = term()

Returns all testcases in the specified suite.

`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) -> TCUserData | {error, Reason}`

Types:

TestDir = string()

Suite = atom()

Case = atom()

TCUserData = [term()]

Reason = term()

Returns any data specified with the tag `userdata` in the list of tuples returned from `Suite:Case/0`.

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.

`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}

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) -> term()`

`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.

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.

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}]}.
```

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) -> term()

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) -> term()

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) -> term()

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) -> term()

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) -> term()

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`).

`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.

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) -> term()

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) -> term()

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) -> term()

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) -> term()

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) -> term()

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) -> term()

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) -> term()

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) -> term()

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) -> term()

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) -> term()

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) -> term()

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) -> term()

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) -> term()

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) -> term()

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) -> term()

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) -> term()

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) -> term()

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) -> term()`

`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) -> term()`

`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) -> term()`

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 = fun()

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 = fun()

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 = fun()

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 = fun()

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() = string()
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

`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

`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

`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.

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

Types:

MgrAgentConfName = atom()

Reason = term()

Removes information added when calling register_agents/2.

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

Types:

MgrAgentConfName = atom()

Reason = term()

Removes information added when calling register_users/2.

`update_usm_users(MgrAgentConfName, UsmUsers) -> ok | {error, Reason}`

Types:

MgrAgentConfName = atom()

UsmUsers = usm_users()

Reason = term()

Alters information added when calling register_usm_users/2.

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)
```

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}]}.
```

`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.

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)

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, DefaultTimeout)*.

`cmd(Connection, Cmd, Timeout) -> term()`

`cmdf(Connection, CmdFormat, Args) -> {ok, Data} | {error, Reason}`

Equivalent to *cmdf(Connection, CmdFormat, Args, DefaultTimeout)*.

`cmdf(Connection, CmdFormat, Args, Timeout) -> term()`

`cont_log(Str, Args) -> term()`

`end_log() -> term()`

`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 = {timeout, Timeout} | repeat | {repeat, N} | sequence | {halt, HaltPatterns} | ignore_prompt
Timeout = 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. If more than one pattern is given, the function returns when the first match is found.

`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 function will always return when a prompt is found, unless the `ignore_prompt` options is used.

The `timeout` option indicates that the function shall return if the telnet client is idle (i.e. if no data is received) for more than `Timeout` milliseconds. Default timeout is 10 seconds.

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.

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

Types:

Connection = connection() (see module ct_telnet)

Data = [string()]

Get all data which has been received by the telnet client since last command was sent.

`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)

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()

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`).

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

Types:

Connection = connection() (see module ct_telnet)

Cmd = string()

Send a telnet command and return immediately.

The resulting output from the command can be read with `get_data/1` or `expect/2/3`.

`sendf(Connection, CmdFormat, Args) -> ok | {error, Reason}`

Types:

Connection = connection() (see module ct_telnet)

CmdFormat = string()

Args = list()

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 talking telnet to 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 talk telnet to the host specified by `HostNameOrIpAddress`, use the interface functions in `ct`, 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,username,password]}}].
```

or

```
ct:require(Name,{unix,[telnet,username,password]}).
```

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).

See also

ct, *ct_telnet*