Common Test

version 1.3

Typeset in LaTeX from SGML source using the DocBuilder-0.9.7 Document System.

Contents

Chapter 1

Common Test User's Guide

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

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

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

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

1.1 Common Test Basics

1.1.1 Introduction

The Common Test framework (CT) is a tool which 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.

For white-box testing of Erlang code, the test programs can of course call Erlang API functions directly. Black-box testing of Erlang code can use Erlang RPC as well as standard O&M interfaces if desired.

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 CT!

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 some CVS-like tool). 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 are functions that are useful for all test suites or for test suites in a specific functional area or subsystem. So as well as the general support libraries provided by the CT framework there might be a need for customized support libraries on a subsystem level.

1.1.4 Scripting Suite and Test Cases

Testing is performed by running test suites (a set 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.

The test suite file must conform to a certain interface which is specified by the CT test server. See the section on writing test suites 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 an HTML log file which is created for the particular test run. Each test case is represented by a row in a table that shows the total execution time, whether the case was successful or if it failed or was skipped, plus a possible user comment. For a failed test case, the reason for termination is printed. The HTML file has a link to the logfile for each test case (which may also be viewed with an HTML browser).

1.1.5 External Interfaces

The CT test server requires some default functions in a test suite. Each suite module should define and export the following functions (most are however optional):

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

suite() Default suite configuration. (Optional)

sequences() Specifies dependencies between test cases. (Optional)

init_per_suite(Conf) Executed before the first test case in a suite. (Optional)

end_per_suite(Conf) Executed after the last test case in a suite. (Optional)

init_per_testcase(TC, Conf) Executed before each test case in the suite. (Optional)

end_per_testcase(TC, Conf) Executed after each test case in the suite. (Optional)

For each test case the CT test server needs these functions:

Testcasename() Returns a key-value list of test case configuration/information. (Optional)

Testcasename(Config) The actual test case function.

1.2 Installation

1.2.1 Unix

Copy the Common Test and Test Server application directories, `common_test-<vsn>` and `test_server-<vsn>`, to a location of your choice. They do not have to be placed among the Erlang applications under the OTP lib directory, nor do they need to have a particular path relative to your test suite modules. In the Common Test directory you find the shell script `install.sh`. Execute this script to generate the Common Test start script `run_test` in the sub directory `common_test-<vsn>/priv/bin`.

`install.sh` takes one input parameter which specifies the absolute path to the top directory of Common Test and Test Server. (This path is inserted in the `run_test` script so that when the script starts Erlang, the Erlang code server will be able to load the Common Test and Test Server application modules). Example (assuming Common Test and Test Server have been placed in `/usr/local/test_tools`):

```
$ 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 for the installation to work properly. Note also that the install script does not e.g. copy files or update environment variables. It only generates the `run_test` script.

If the directories are later moved, make sure to run `install.sh` again or edit the `run_test` script (Bourne shell) manually.

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

1.2.2 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 for installing Common Test on Unix above.

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. In this case you do not need to install Common Test (i.e. no need to generate the `run_test` script). Simply copy the `common_test-<vsn>` and `test_server-<vsn>` directories to a location of your choice. They do not have to be placed among the Erlang applications under the OTP lib directory, nor do they need to have a particular path relative to your test suite modules.

When you start the Erlang node on which you will be running Common Test, make sure the Common Test and Test Server `ebin` directories are included in the Erlang code server path (so the application modules can be loaded). If you do copy the application directories to the OTP lib directory, there is no need to explicitly update the code server path as the code server will be able to locate the modules automatically.

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:

- 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
- Tracing of line numbers in the test suite, i.e. if a test case fails, the last 10 executed line numbers are displayed

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

The CT application also includes other modules named `ct_<something>` that provide support for the use of communication mechanisms such as rpc, snmp, ftp, telnet etc in test suites.

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 function in CT will not be able to locate the modules (per default).

`ct.hrl` shall be included in all test suite files.

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

1.3.3 Init and end per suite

Each test suite module may contain the functions `init_per_suite/1` and `end_per_suite/1`.

If it exists, `init_per_suite` is called as the first testcase of the suite. It typically contains initiation which is common for all test cases in the suite, and which are only to be performed once.

`end_per_suite` is called as the last test case of the suite. This function should clean up after `init_per_suite`.

The argument to `init_per_suite` is `Config`, the same as the argument to all test cases. `init_per_suite` can modify this parameter with information that the other test cases need.

If `init_per_suite` fails, all test cases in the test suite will be skipped, including `end_per_suite`

It is possible in `end_per_suite` to check if the last executed 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`}, or {`skipped`,`Reason`}.

1.3.4 Init and end per test case

Each test suite module can contain the functions `init_per_testcase/2` and `end_per_testcase/2`.

If it exists, `init_per_testcase` is called before each test case in the suite. It typically contains initiation which must be done for each test case.

`end_per_testcase/2` is called after each test case is completed, giving a possibility to clean up.

The first argument to these functions is the name of the test case. This can be used to do individual initiation and cleanup for each test cases.

The second argument is called `Config`. `init_per_testcase/2` may modify this parameter or just return it as is. Whatever is retuned by `init_per_testcase/2` is given as `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 [page 32] tuple).

If `init_per_testcase` crashes, the test case itself is skipped and `end_per_testcase` is never called.

1.3.5 Test cases

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

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

Using many small test cases tend to result in extra and often duplicated code as well as slow test execution because of large overhead for initializations and cleanups. Lots of duplicated code results in high maintenance cost and bad readability.

Larger test cases make it harder to tell what went wrong if it fails, and large portions of test code will be skipped if a specific part fails. Also, readability and maintainability suffers when test cases become too extensive.

The test case function takes one argument, `Config`, which contains configuration information such as `data_dir` and `priv_dir`. See Data and Private Directories [page 7] for more information about these.

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. A configuration file should contain data related to the SUT (system under test). These two types of config data are handled differently!

All `Config` items can be extracted using the `?config` macro, e.g `PrivDir = ?config(priv_dir,Config)`.

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

If the test case returns the tuple `{comment,Comment}`, `Comment` is printed out in the overview log (this is 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 take. 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 [page 21] 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, start by adding a normal require-tuple to the list for the variable in question and then let the key-value definition for the variable follow. The definition should have the same form as in the configuration file. Example:

```
my_testcase() ->
    [{require, unix_telnet, {unix, [telnet, username, password]}},
     {unix_telnet, {unix, [{telnet, "durin.du.uab"},
                           {username, "alladin"},
                           {password, "sesame"}]}}].
```

Note:
Giving a required variable a default value can result in that a test case is always run. For many types of configuration values this is not a desired behavior.

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:

```
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 default data

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 defaults list are:

- stylesheet, see HTML Style Sheets [page 18].
- userdata, see Test case info function [page 6].
- silent_connections, see Silent Connections [page 21].

Example:

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

1.3.8 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/".

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`. If you do need to use the current working directory, you must set it explicitly with `file:set_cwd/1` for each individual test case before use. (The Common Test server sets current working directory to the test case log directory at the start of every case).

1.3.9 Execution environment

Each test case, including `init_per_testcase` and `end_per_testcase` is executed by a dedicated Erlang process. The process is spawned when the test case starts, and terminated when the test case is finished.

`init_per_suite` and `end_per_suite` are separate test cases and will execute on their own 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 by the `suite/0` function.

1.3.10 Illegal dependencies

Even though it is highly efficient to write test suites with the Common Test framework, there will be mistakes in the test suites. Noted below are some of the more frequent dependency mistakes from our 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. If the current directory has to be set to something in particular, use `file:set_cwd/1` to set it. And use the `data_dir` and `priv_dir` to locate data and scratch files.
- Depending on the Clearcase (file version control tool) 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:
  There is no way of telling in which order the test cases are going to be run, so a test case can't depend on a server being started by a test case that runs "before". This has to be so for several reasons:
  The user may specify the order at will, and maybe some particular order is better suited sometimes. Secondly, if the user just specifies a test directory, 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 `unix:cmd` or `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 following test cases:
  Test cases should restore as much of the execution environment as possible, so that the following 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 consists of a set of test cases. Each test case is implemented as an Erlang function. An Erlang module implementing one or more test cases is called a test suite. One or more test suites are stored in a test directory.

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 [page 15].
- 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 is not run.

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

1.4.3 Definition of terms

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.

major log file An overview log file for one or more test suites.

minor log file A log file for one particular test case.

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

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

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.

1.5 Examples

1.5.1 Test suite

The example test suite shows some tests of an HTTP client that uses a proxy.

```
-module(httpc_proxy_SUITE).

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

-include("ct.hrl").

-define(URL, "http://www.erlang.org").
-define(PROXY, "www-proxy.ericsson.se").
-define(PROXY_PORT, 8080).

%%--------------------------------------------------------------------
%% Test server callback functions
%%--------------------------------------------------------------------

%%--------------------------------------------------------------------
%% Function: suite() -> DefaultData
%% DefaultData: [tuple()]
%% Description: Require variables and set default values for the suite
%%--------------------------------------------------------------------
suite() -> [{timetrap,{minutes,1}}].

%%--------------------------------------------------------------------
%% Function: init_per_suite(Config) -> Config
%% Config: [tuple()]
%%   A list of key/value pairs, holding the test case configuration.
%% Description: Initiation before the whole 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) ->
    application:start(inets),
    http:set_options([{proxy, {{?PROXY, ?PROXY_PORT}, ["localhost"]}}]),
    Config.

%%--------------------------------------------------------------------
%% Function: end_per_suite(Config) -> _
%% Config: [tuple()]
%%   A list of key/value pairs, holding the test case configuration.
%% Description: Cleanup after the whole suite
%%--------------------------------------------------------------------
end_per_suite(_Config) ->
    application:stop(inets),
    ok.
%%--------------------------------------------------------------------
```

```
%% Function: all() -> TestCases
%% TestCases: [Case]
%% Case: atom()
%%   Name of a test case.
%% Description: Returns a list of all test cases in this test suite
%%--------------------------------------------------------------------
all() ->
    [options, head, get, trace].

%%--------------------------------------------------------------------
%% Test cases starts here.
%%--------------------------------------------------------------------

options() ->
    [{userdata,[{doc,"Perform an OPTIONS request that goes through a proxy."}]}].

options(_Config) ->
    {ok, {{_,200,_}, Headers, _}} =
        http:request(options, {?URL, []}, [], []),
    case lists:keysearch("allow", 1, Headers) of
        {value, {"allow", _}} ->
            ok;
        _ ->
            ct:fail(http_options_request_failed)
    end.

head() ->
     [{userdata,[{doc,"Perform a HEAD request that goes through a proxy."}]}].

head(_Config) ->
    {ok, {{_,200, _}, [_ | _], []}} =
        http:request(head, {?URL, []}, [], []).

get() ->
    [{userdata,[{doc, "Perform a GET request that goes through a proxy."}]}].

get(_Config) ->
    {ok, {{_,200,_}, [_ | _], Body = [_ | _]}} =
        http:request(get, {?URL, []}, [], []),
    check_body(Body).

trace() ->
    [{userdata,[{doc, "Perform a TRACE request that goes through a proxy."}]}].

trace(_Config) ->
    {ok, {{_,200,_}, [_ | _], "TRACE /" ++ _}} =
        http:request(trace, {?URL, []}, [], []),
    ok.

%%--------------------------------------------------------------------
%% Internal functions
%%--------------------------------------------------------------------
```

```
check_body(Body) ->
     case string:rstr(Body, "\html>") of
        0 ->
            ct:fail(did_not_receive_whole_body);
        _ ->
            ok
    end.
```

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 target nodes and other variables.
- 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.

1.6.3 Running tests from the UNIX command line

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

- `run_test -config <configfilenames> -dir <dirs>`
- `run_test -config <configfilenames> -suite <suiteswithfullpath>`
- `run_test -config <configfilenames> -suite <suitewithfullpath> -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
```

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).
- `-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 [page 23]).
- `-event_handler <event_handlers>`, to install event handlers [page 27].
- `-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).

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-2.0.0.3&'
```

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. For the VTS mode to work properly with netscape, make sure the `netscape` program in your path starts version 7!

1.6.5 Running tests from the Erlang shell

Common Test provides an Erlang API for running tests. For documentation, please see the `ct` manual page.

1.6.6 Running interactive shell mode

You can start an Erlang shell with the necessary paths and with Common Test running in an interactive mode with the `run_test` script and the `-shell` option:

- `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.

From the interactive mode all test case support functions can be executed directly from the erlang shell. This is an experimentation mode useful during test suite development and debugging.

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/[1,2]`. This is equivalent to a `require` statement in the Test Suite Default Data [page 7] or in the Test Case Info Function [page 6].

Example:

```
> ct:require(a,{unix,[telnet]}).
ok
> ct:cmd(a,"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`.

If you for some reason want to exit the interactive mode, use 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 previous state is not restored.

1.6.7 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 [page 23] chapter) and specify event_handler plugins (see the Event Handling [page 27] chapter).

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.

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 [page 27] chapter.

Config terms:

```
{node, NodeAlias, Node}.

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

{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()
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}`

1.6.8 Log files

As the execution of the test suites go on, 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 screen with details about/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 somewhere other than `standard_io`.

This is enough for the operator to know, and if he wants to dig in deeper into a specific test case result, he can do so by following the links in the HTML presentation to 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.

A detailed report of the test run is stored in the major logfile (textual log file). This includes test suite and test case names, execution time, the exact reason for failure etc. There is an HTML log file that corresponds to this textual file. The HTML file is a summary which gives a better 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.

1.6.9 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>
```

1.6.10 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.

1.6.11 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 or a test plant. 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 config file can contain any number of elements of the type:

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

where

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

1.7.3 Reading config values

From inside a test suite, one can *require* a variable to exist before a test case can be run, and *read* the value of a variable.

`require` is a statement that can be part of the test suite default data [page 7] or test case info function [page 6]. If the required variable is not available, the test is skipped. There is also a function `ct:require/[1,2]` which can be called from inside a test suite in order to check if a specific variable is available. The return value from the function must then be checked and appropriate actions taken explicitly by the test case.

A `require` statement 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 [page 50].

To read the value of a config variable, use the function `get_config/[1,2]` which is also described in the reference manual for ct [page 50].

1.7.4 Examples

A config file for usning telnet to communicate with a unix host could look like this:

```
{unix,[{telnet,"durin.du.uab"},
       {username,"someuser"},
       {password,"somepassword"}]}.
```

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 [page 15]).

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).

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 [page 15] 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 [page 15] 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 specfies 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 [page 15] 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 session execution. It is reported e.g. when a test case starts and stops, what the current count of succeeded, 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 event notifications, take some action or 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 actually 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 plugged in by means of test specification [page 15] 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 modules (each having gen_event behaviour). An event handler module must be precompiled and its location must be specified in the Erlang runtime system code path. 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 data specific for the particular event.

Events:

```
#event.name = test_start
#event.data = {StartTime,LogDir}
StartTime = {date(), time()}
LogDir = string()

#event.name = test_done
#event.data = EndTime
EndTime = {date(), time()}

#event.name = start_make
#event.data = Dir
Dir = string()

#event.name = finished_make
#event.data = Dir
Dir = string()

#event.name = tc_start
#event.data = {Suite,Case}
Suite = atom()
Case = atom()

#event.name = tc_done
#event.data = {Suite,Case,Result}
Suite = atom()
Case = atom()
Result = ok | {skipped,Reason} | {failed,Reason}
Reason = term()

#event.name = tc_user_skip
```

```
#event.data = {Suite,Case,Comment}
Suite = atom()
Case = atom()
Comment = string()

#event.name = tc_auto_skip
#event.data = {Suite,Case,Comment}
Suite = atom()
Case = atom()
Comment = string()

#event.name = test_stats
#event.data = {Ok,Failed,Skipped}
Ok = Failed = Skipped = integer()

#event.name = start_logging
#event.data = CtRunDir
CtRunDir = string()

#event.name = stop_logging
#event.data = []

#event.name = start_write_file
#event.data = FullNameFile
FullNameFile = string()

#event.name = finished_write_file
#event.data = FullNameFile
FullNameFile = string()
```

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.

Besides the event_handler test specification terms, it is also possible to install event handlers by means of the `run_test` flag `-event_handler`, e.g:

```
$ run_test -dir my_testobj -event_handler my_evh1 my_evh2
```

Note that it is not posible 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([{dir,"my_testobj"},{event_handler,[my_evh1,{my_evh2,[node()]}]}]).
```

This will install two event handlers for the `my_testobj` 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.

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 declared by means of the function `sequences/0`. This function should return a list of tuples on the format: `{SeqTag,TestCases}`. `SeqTag` is an atom that identifies one particular sequence. `TestCases` is a list of test case names. Sequences must be included in the list that `all/0` returns. They are declared as: `{sequence,SeqTag}`.

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

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

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

```
all() -> [{sequence,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 sequence tuples may be specified. Example:

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

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

Note:
It is not possible to execute a test case which is part of a sequence as a regular (stand alone) test case. It is also not possible to use the same test case in multiple sequences. Remember that you can always work around these limitations if necessary by means of help functions common for the test cases in question.

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 loadmodules 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.

Common Test Reference Manual

Short Summaries

- Application common_test [page 44] – A framework for automatic testing of a variety of target nodes
- Application run_test [page 48] – Shell script used for starting Common Test from the Unix command line.
- Erlang Module ct [page 50] – Main user interface for the Common Test framework.
- Erlang Module ct_cover [page 58] – Common Test Framework code coverage support module.
- Erlang Module ct_ftp [page 59] – Common Test specific layer on top of OTP ftp clinet ftp.erl.
- Erlang Module ct_master [page 62] – Distributed test execution control for Common Test.
- Erlang Module ct_rpc [page 65] – Common Test specific layer on Erlang/OTP rpc.
- Erlang Module ct_snmp [page 68] – Common Test specific layer on top of the OTPs snmp.
- Erlang Module ct_telnet [page 73] – Common Test specific layer on top of telnet client ct_telnet_client.erl.
- Erlang Module unix_telnet [page 78] – Callback module for ct_telnet for talking telnet to a unix host.

common_test

The following functions are exported:

- `Module:all() -> TestCases | {skip,Reason}`
  [page 44] Returns the list of all test cases in the module.
- `Module:sequences() -> Sequences`
  [page 45] Returns a list of test cases that depend on each other.
- `Module:suite() -> [Info]`
  [page 45] Function for providing default test suite data.
- `Module:init_per_suite(Config) -> NewConfig | {skip,Reason} | {skip_and_save,Reason,Config}`
  [page 45] Test suite initialization.

- `Module:end_per_suite(Config) -> void() | {save_config,Config}`
  [page 45] Test suite finalization.
- `Module:init_per_testcase(TestCase, Config) -> NewConfig | {skip,Reason}`
  [page 46] Test case initialization.
- `Module:end_per_testcase(TestCase, Config) -> void() | {save_config,Config}`
  [page 46] Test case finalization.
- `Module:testcase() -> [Info]`
  [page 46] Test case info function.
- `Module:testcase(Config) -> ok | {skip,Reason} | {comment,Comment} | {save_config,Config} | {skip_and_save,Reason,Config} | exit()`
  [page 47] A test case

run_test

No functions are exported.

ct

The following functions are exported:

- `comment(Comment) -> void()`
  [page 50] Print the given Comment in the comment field of the table on the test suite result page.
- `fail(Reason) -> void()`
  [page 50] Terminate a test case with the given error Reason.
- `get_config(Required) -> Value`
  [page 51] Equivalent to get_config(Required, undefined).
- `get_config(Required, Default) -> Value`
  [page 51] Get the value of config data.
- `get_status() -> TestStatus | {error, Reason}`
  [page 51] Returns status of ongoing tests.
- `get_target_name(Handle) -> {ok, TargetName} | {error, Reason}`
  [page 51] Return the name of the target that the given connection belongs to.
- `install(Opts) -> ok | {error, Reason}`
  [page 51] Install config files and event handlers.
- `listenv(Telnet) -> [Env]`
  [page 52] Performs the listenv command on the given telnet connection and returns the result as a list of Key-Value pairs.
- `log(Format) -> ok`
  [page 52] Equivalent to log(default, Format, []).
- `log(X1, X2) -> ok`
  [page 52] Equivalent to log(Category, Format, Args).
- `log(Category, Format, Args) -> ok`
  [page 52] Printout from a testcase to the log.
- `pal(Format) -> ok`
  [page 52] Equivalent to pal(default, Format, []).

- `pal() -> term()`
  [page 53] Equivalent to pal(Category, Format, Args).
- `pal(Category, Format, Args) -> ok`
  [page 53] Print and log from a testcase.
- `parse_table(Data) -> {Heading, Table}`
  [page 53] Parse the printout from an SQL table and return a list of tuples.
- `print(Format) -> ok`
  [page 53] Equivalent to print(default, Format, []).
- `print() -> term()`
  [page 53] Equivalent to print(Category, Format, Args).
- `print(Category, Format, Args) -> ok`
  [page 53] Printout from a testcase to the console.
- `require(Required) -> ok | {error, Reason}`
  [page 53] Check if the required configuration is available.
- `require(Name, Required) -> ok | {error, Reason}`
  [page 54] Check if the required configuration is available, and give it a name.
- `run(TestDirs) -> Result`
  [page 54] Run all testcases in all suites in the given directories.
- `run(TestDir, Suite) -> Result`
  [page 54] Run all testcases in the given suite.
- `run(TestDir, Suite, Cases) -> Result`
  [page 55] Run the given testcase(s).
- `run_test(Opts) -> Result`
  [page 55] Run tests as specified by the combination of options in Opts.
- `run_testspec(TestSpec) -> Result`
  [page 55] Run test specified by TestSpec.
- `start_interactive() -> ok`
  [page 56] Start CT in interactive mode.
- `step(TestDir, Suite, Case) -> Result`
  [page 56] Step through a test case with the debugger.
- `stop_interactive() -> ok`
  [page 56] Exit the interactive mode.
- `testcases(TestDir, Suite) -> Testcases | {error, Reason}`
  [page 56] Returns all testcases in the specified suite.
- `userdata(TestDir, Suite) -> SuiteUserData | {error, Reason}`
  [page 56] 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}`
  [page 57] Returns any data specified with the tag userdata in the list of tuples returned from Suite:Case/0.

ct_cover

The following functions are exported:

- `add_nodes(Nodes) -> {ok, StartedNodes} | {error, Reason}`
  [page 58] Add nodes to current cover test (only works if cover support is active!).
- `remove_nodes(Nodes) -> ok | {error, Reason}`
  [page 58] Remove nodes from current cover test.

ct_ftp

The following functions are exported:

- `cd(Connection, Dir) -> ok | {error, Reason}`
  [page 59] Change directory on remote host.
- `close(Connection) -> ok | {error, Reason}`
  [page 59] Close the FTP connection.
- `delete(Connection, File) -> ok | {error, Reason}`
  [page 59] Delete a file on remote host.
- `get(Name, RemoteFile, LocalFile) -> ok | {error, Reason}`
  [page 59] Open a ftp connection and fetch a file from the remote host.
- `ls(Connection, Dir) -> {ok, Listing} | {error, Reason}`
  [page 60] List the directory Dir.
- `open(Name) -> {ok, Handle} | {error, Reason}`
  [page 60] Open an FTP connection to the specified node.
- `put(Name, LocalFile, RemoteFile) -> ok | {error, Reason}`
  [page 60] Open a ftp connection and send a file to the remote host.
- `recv(Connection, RemoteFile) -> ok | {error, Reason}`
  [page 60] Fetch a file over FTP.
- `recv(Connection, RemoteFile, LocalFile) -> ok | {error, Reason}`
  [page 60] Fetch a file over FTP.
- `send(Connection, LocalFile) -> ok | {error, Reason}`
  [page 61] Send a file over FTP.
- `send(Connection, LocalFile, RemoteFile) -> ok | {error, Reason}`
  [page 61] Send a file over FTP.
- `type(Connection, Type) -> ok | {error, Reason}`
  [page 61] Change file transfer type.

ct_master

The following functions are exported:

- `abort() -> ok`
  [page 62] Stops all running tests.
- `abort(Nodes) -> ok`
  [page 62] Stops tests on specified nodes.
- `progress() -> [{Node, Status}]`
  [page 62] Returns test progress.
- `run(TestSpecs) -> ok`
  [page 62] Equivalent to run(TestSpecs, false, [], []).
- `run(TestSpecs, InclNodes, ExclNodes) -> ok`
  [page 62] Equivalent to run(TestSpecs, false, InclNodes, ExclNodes).
- `run(TestSpecs, AllowUserTerms, InclNodes, ExclNodes) -> ok`
  [page 63] Tests are spawned on the nodes as specified in TestSpecs.
- `run_on_node(TestSpecs, Node) -> ok`
  [page 63] Equivalent to run_on_node(TestSpecs, false, Node).

- `run_on_node(TestSpecs, AllowUserTerms, Node) -> ok`
  [page 63] Tests are spawned on Node according to TestSpecs.
- `run_test(Node, Opts) -> ok`
  [page 63] Tests are spawned on Node using ct:run_test/1.

ct_rpc

The following functions are exported:

- `app_node(App, Candidates) -> NodeName`
  [page 65] From a set of candidate nodes determines which of them is running the application App.
- `app_node(App, Candidates, FailOnBadRPC) -> NodeName`
  [page 65] 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`
  [page 65] 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.
- `call(Node, Module, Function, Args) -> term() | {badrpc, Reason}`
  [page 66] Same as call(Node, Module, Function, Args, infinity).
- `call(Node, Module, Function, Args, TimeOut) -> term() | {badrpc, Reason}`
  [page 66] Evaluates apply(Module, Function, Args) on the node Node.
- `call(Node, Module, Function, Args, TimeOut, Cookie) -> term() | {badrpc, Reason}`
  [page 66] Evaluates apply(Module, Function, Args) on the node Node.
- `cast(Node, Module, Function, Args) -> ok`
  [page 66] Evaluates apply(Module, Function, Args) on the node Node.
- `cast(Node, Module, Function, Args, Cookie) -> ok`
  [page 67] Evaluates apply(Module, Function, Args) on the node Node.

ct_snmp

The following functions are exported:

- `get_next_values(Agent, Oids, ConfName) -> SnmpReply`
  [page 70] Issues a synchronous snmp get next request.
- `get_values(Agent, Oids, ConfName) -> SnmpReply`
  [page 70] Issues a synchronous snmp get request.
- `load_mibs(Mibs) -> ok | {error, Reason}`
  [page 70] Load the mibs into the agent 'snmp_master_agent'.
- `register_agents(ConfName, ManagedAgents) -> ok | {error, Reason}`
  [page 70] Explicitly instruct the manager to handle this agent.
- `register_users(ConfName, Users) -> ok | {error, Reason}`
  [page 71] Register the manager entity (=user) responsible for specific agent(s).
- `register_usm_users(ConfName, UsmUsers) -> ok | {error, Reason}`
  [page 71] Explicitly instruct the manager to handle this USM user.

- `set_info(Config) -> [{Agent, OldVarsAndVals, NewVarsAndVals}]`
  [page 71] Returns a list of all successful set requests performed in the test case in reverse order.
- `set_values(Agent, VarsAndVals, ConfName, Config) -> SnmpReply`
  [page 71] Issues a synchronous snmp set request.
- `start(Config, ConfName) -> ok`
  [page 71] Starts an snmp manager and/or agent.
- `stop(Config) -> ok`
  [page 72] Stops the snmp manager and/or agent removes all files created.
- `unregister_agents(ConfName) -> ok | {error, Reason}`
  [page 72] Removes information added when calling register_agents/2.
- `unregister_users(ConfName) -> ok | {error, Reason}`
  [page 72] Removes information added when calling register_users/2.
- `update_usm_users(ConfName, UsmUsers) -> ok | {error, Reason}`
  [page 72] Alters information added when calling register_usm_users/2.

ct_telnet

The following functions are exported:

- `close(Connection) -> ok | {error, Reason}`
  [page 74] Close the telnet connection and stop the process managing it.
- `cmd(Connection, Cmd) -> {ok, Data} | {error, Reason}`
  [page 74] Equivalent to cmd(Connection, Cmd, DefaultTimeout).
- `cmd(Connection, Cmd, Timeout) -> {ok, Data} | {error, Reason}`
  [page 74] Send a command via telnet and wait for prompt.
- `cmdf(Connection, CmdFormat, Args) -> {ok, Data} | {error, Reason}`
  [page 74] Equivalent to cmdf(Connection, CmdFormat, Args, DefaultTimeout).
- `cmdf(Connection, CmdFormat, Args, Timeout) -> {ok, Data} | {error, Reason}`
  [page 74] Send a telnet command and wait for prompt (uses a format string and list of arguments to build the command).
- `cont_log() -> term()`
  [page 74]
- `end_log() -> term()`
  [page 74]
- `expect(Connection, Patterns) -> term()`
  [page 74] Equivalent to expect(Connections, Patterns, []).
- `expect(Connection, Patterns, Opts) -> {ok, Match} | {ok, MatchList, HaltReason} | {error, Reason}`
  [page 74] Get data from telnet and wait for the expected pattern.
- `get_data(Connection) -> {ok, Data} | {error, Reason}`
  [page 76] Get all data which has been received by the telnet client since last command was sent.
- `open(Name) -> {ok, Handle} | {error, Reason}`
  [page 76] Equivalent to open(Name, telnet).
- `open(Name, ConnType) -> {ok, Handle} | {error, Reason}`
  [page 76] Open a telnet connection to a node.

- `open(Name, ConnType, TargetMod) -> {ok, Handle} | {error, Reason}`
  [page 76] Equivalent to open(Name, ConnType, TargetMod, []).
- `open(Name, ConnType, TargetMod, Extra) -> {ok, Handle} | {error, Reason}`
  [page 76] Open a telnet connection to the specified target.
- `send(Connection, Cmd) -> ok | {error, Reason}`
  [page 76] Send a telnet command and return immediately.
- `sendf(Connection, CmdFormat, Args) -> ok | {error, Reason}`
  [page 77] Send a telnet command and return immediately (uses a format string and a list of arguments to build the command).

unix_telnet

No functions are exported.

Common Test

Application

The *Common Test* framework is an environment for writing and executing automatic and semi-automatic test cases. The framework is based on the underlying implementation as is used with great success in the Visual Test Server (AXD301) and by Erlang/OTP. The main function in this framework is the test server which runs all the test cases.

In brief the test server supports:

- Running multiple test suites
- Logging of the events in a test suite, on both suite and case levels
- HTML presentation of test suite results
- HTML presentation of test suite code
- Support functions for test suite authors
- Step by step execution

The following sections describes the callback functions the test server expects to find in a test suite. For more details see Common Test User's Guide. [page 4]

TEST CASE CALLBACK FUNCTIONS

The following functions define the callback interface for a test suite.

Exports

`Module:all() -> TestCases | {skip,Reason}`

Types:

- TestCases = [atom() | {sequence,SeqName}]
- Reason = term()
- SeqName = atom()

MANDATORY

This function must return the list of all test cases in the test suite module. Each test case is represented by an atom, the name of the test case function.

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 sequences, see Dependencies between Test Cases and Suites [page 33] in the User's Guide.

`Module:sequences() -> Sequences`

Types:

- Sequences = [{SeqName,Testcases}]
- SeqName = atom()
- Testcases = [atom()]

OPTIONAL

See Dependencies between Test Cases and Suites [page 33] 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

Use this function to set default data for the test suite.

`Module:init_per_suite(Config) -> NewConfig | {skip,Reason} | {skip_and_save,Reason,Config}`

Types:

- Config = NewConfig = [{Key, Value}]
- Key = atom()
- Value = term()
- Reason = term()

OPTIONAL

This function is called as the first test case 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 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 [page 32] in the User's Guide.

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

Types:

- Config = [{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`. For information on `save_config`, please see Dependencies between Test Cases and Suites [page 32] in the User's Guide.

`Module:init_per_testcase(TestCase, Config) -> NewConfig | {skip,Reason}`

Types:

- 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() | {save_config,Config}`

Types:

- Config = [{Key, Value}]
- Key = atom()
- Value = term()

OPTIONAL

This function is called after each test case, and can be used to clean up whatever the test case has done. The return value is ignored. For information on `save_config`, please see Dependencies between Test Cases and Suites [page 32] 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 shall return a list of tagged tuples that specify various properties regarding the 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` and `end_per_testcase` 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.

Apart from the above mentioned tags, there is no limitation for which tags that can be specified in the test case info function.

`Module:testcase(Config) -> ok | {skip,Reason} | {comment,Comment} | {save_config,Config} | {skip_and_save,Reason,Config} | exit()`

Types:

- Config = [{Key, Value}]
- Key = atom()
- Value = term()

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 someting fails, make sure the process crashes or call `ct:fail/[0,1]` (which also will cause the 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, it is considered a success.

For information on `save_config` and `skip_and_save`, please see Dependencies between Test Cases and Suites [page 32] in the User's Guide.

The run_test shell script

Application

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
[-step]
[-config ConfigFile1 ConfigFile2 .. ConfigFileN]
[-logdir LogDir]
[-silent_connections [ConnType1 ConnType2 .. ConnTypeN]]
[-stylesheet CSSFile]
[-cover CoverCfgFile]
[-event_handler EvHandler1 EvHandler2 .. EvHandlerN]
[-repeat N [-force_stop]] |
[-duration HHMMSS [-force_stop]] |
[-until [YYMoMoDD]HHMMSS [-force_stop]]
```

Run tests using test specification

```
run_test -spec TestSpec1 TestSpec2 .. TestSpecN
[-config ConfigFile1 ConfigFile2 .. ConfigFileN]
[-logdir LogDir]
[-allow_user_terms]
[-silent_connections [ConnType1 ConnType2 .. ConnTypeN]]
[-stylesheet CSSFile]
[-cover CoverCfgFile]
[-event_handler EvHandler1 EvHandler2 .. EvHandlerN]
[-repeat N [-force_stop]] |
[-duration HHMMSS [-force_stop]] |
[-until [YYMoMoDD]HHMMSS [-force_stop]]
```

Run tests in web based GUI

```
run_test -vts [-browser Browser]
[-config ConfigFile1 ConfigFile2 .. ConfigFileN]
[-dir TestDir1 TestDir2 .. TestDirN] |
[-suite Suite [-case Case]]
```

Refresh the HTML index files

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

Run CT in interactive mode

```
run_test -shell [-config ConfigFile1 ConfigFile2 ... ConfigFileN]
```

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 [page 12] 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

`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).

`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) [page 51].

`get_config(Required, Default) -> Value`

Types:

- Required = KeyOrName | {KeyOrName, SubKey}
- KeyOrName = atom()
- SubKey = atom()
- Default = term()
- Value = term() | Default

Get the value of config data.

This function returns the value of the requested config element.

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 you want to access a config variable which has been given a name by require/2, the name must be used instead of the key when reading the value:

```
require(myhost,unix) -> ok
get_config(myhost,Default) -> [{telnet,IpAddr}, {username,Username},
{password,Password}]
```

See also: get_config/1 [page 51], require/1 [page 54], require/2 [page 54].

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

Types:

- TestDir = term()
- Reason = term()

Returns status of ongoing tests.

`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}
- ConfigFiles = [ConfigFile]
- ConfigFile = string()
- Modules = [atom()]

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, []) [page 52].

`log(X1, X2) -> ok`

Types:

- X1 = Category | Format
- X2 = Format | Args

Equivalent to log(Category, Format, Args) [page 52].

`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, []) [page 53].

`pal() -> term()`

Equivalent to pal(Category, Format, Args) [page 53].

`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, []) [page 53].

`print() -> term()`

Equivalent to print(Category, Format, Args) [page 53].

`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 [page 51], get_config/2 [page 51], require/2 [page 54].

`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 marked as allocated. An allocated element can only be used if the correct name is given. This means that to read the value of the element with `get_config/1,2`, you need to provide the `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 must 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 [page 51], get_config/2 [page 51], require/1 [page 54].

`run(TestDirs) -> Result`

Types:

- TestDirs = TestDir | [TestDir]

Run all testcases in all suites in the given directories.

See also: run/3 [page 55].

`run(TestDir, Suite) -> Result`

Run all testcases in the given suite.

See also: run/3 [page 55].

`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} | {spec, TestSpecs} | {allow_user_terms, Bool} | {logdir, LogDir} | {silent_connections, Conns} | {cover, CoverSpecFile} | {event_handler, EventHandlers} | {repeat, N} | {duration, DurTime} | {until, StopTime} | {force_stop, Bool}
- CfgFiles = [string()] | string()
- TestDirs = [string()] | string()
- Suites = [string()] | string()
- Cases = [atom()] | atom()
- TestSpecs = [string()] | string()
- LogDir = string()
- EventHandlers = EH | [EH]
- EH = atom() | {atom(), InitArgs} | {[atom()], InitArgs}
- InitArgs = [term()]
- Conns = all | [atom()]
- CoverSpecFile = string()
- N = integer()
- DurTime = string(HHMMSS)
- StopTime = string(YYMoMoDDHHMMSS) | string(HHMMSS)
- 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(a,{unix,[telnet]}).
ok
> ct:cmd(a,"ls").
{ok,["ls","file1 ...",...]}
```

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

Types:

- Case = atom()

Step through a test case with the debugger.

See also: run/3 [page 55].

`stop_interactive() -> ok`

Exit the interactive mode.

See also: start_interactive/0 [page 56].

`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_cover

Erlang Module

Common Test Framework code coverage support module.

This module exports help functions for performing code coverage analysis.

Exports

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

Types:

- Nodes = [atom()]
- StartedNodes = [atom()]
- Reason = cover_not_running | not_main_node

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

`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

Common Test specific layer on top of OTP ftp clinet ftp.erl

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

Types:

- Name = target_name()
- RemoteFile = string()
- LocaFile = 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 [page 60].

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

Types:

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

List the directory Dir.

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

Types:

- Name = target_name() (see module ct)
- Handle = handle()

Open an FTP connection to the specified node.

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

Types:

- Name = target_name()
- LocaFile = 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 [page 61].

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

Types:

- Connection = connection()

- RemoteFile = string()
- LocaFile = 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 [page 61].

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

Types:

- Connection = connection()
- LocaFile = 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_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, [], []) [page 63].

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

Types:

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

Equivalent to run(TestSpecs, false, InclNodes, ExclNodes) [page 63].

`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) [page 63].

`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_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 specific layer on top of the OTPs snmp

Application to make snmp configuration easier for the test case writer. Many test cases can use default values for everything and then no snmp-configuration files needs to be supplied at all. When it is necessary to change some configuration it can be done for the subset of snmp-configuration files that are relevant, and still all this can be put in to the common-test configuration file or for the more specialized configuration parameters a "simple snmp-configuration file" can be placed in the test suites data directory. ct_snmp will also perform a type check on all supplied configuration. In the manager case the common_test application also will keep track of some manager information so that the test case write does not have to keep track of as much input parameters as if using the OTPs snmp manager directly.

The following 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_con
       {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 ConfName parameter in the functions should be the name you allocated in your test suite using require statement. Example:

```
suite() -> [{require, ConfName,{snmp,[users, managed_agents]}}].
```

or

```
ct:require(ConfName,{snmp,[users, managed_agents]}).
```

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. Ex for the getting the erlang node name from the erlNodeTable in the OTP-MIB

```
Oid = ?erlNodeEntry ++ [?erlNodeName, 1]
```

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
```

```
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') |

var_and_val() = {oid(), value_type(), value()}
varbind() = term()
varbinds() = [varbind()]
varsandvals() = [var_and_val()]
```

Exports

`get_next_values(Agent, Oids, ConfName) -> SnmpReply`

Types:

- Agent = agent_name()
- Oids = oids()
- ConfName = atom()
- SnmpReply = snmpreply()

Issues a synchronous snmp get next request.

`get_values(Agent, Oids, ConfName) -> SnmpReply`

Types:

- Agent = agent_name()
- Oids = oids()
- ConfName = 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(ConfName, ManagedAgents) -> ok | {error, Reason}`

Types:

- ConfName = atom()
- ManagedAgents = [agent()]

- Reason = term()

Explicitly instruct the manager to handle this agent. Corresponds to making an entry in agents.conf

`register_users(ConfName, Users) -> ok | {error, Reason}`

Types:

- ConfName = 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(ConfName, UsmUsers) -> ok | {error, Reason}`

Types:

- ConfName = 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, ConfName, Config) -> SnmpReply`

Types:

- Agent = agent_name()
- Oids = oids()
- ConfName = atom()
- Config = [{Key, Value}]
- SnmpReply = snmpreply()

Issues a synchronous snmp set request.

`start(Config, ConfName) -> ok`

Types:

- Config = [{Key, Value}]
- Key = atom()

- Value = term()
- ConfName = atom()

Starts an snmp manager and/or agent. In the manager case also registrations of users and agents as specified by the configuration <ConfName> 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 registerd 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 to load mibs into the agent.

`stop(Config) -> ok`

Types:

- Config = [{Key, Value}]
- Key = atom()
- Value = term()
- ConfName = atom()

Stops the snmp manager and/or agent removes all files created.

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

Types:

- ConfName = atom()
- Reason = term()

Removes information added when calling register_agents/2.

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

Types:

- ConfName = atom()
- Reason = term()

Removes information added when calling register_users/2.

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

Types:

- ConfName = 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 ct_telnet may be used specifically with unix hosts.)

The rx driver used by ct_telnet for handling regular expressions is currently only ported to Unix and Linux and will not work on Windows!

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)
```

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}]}.
```

`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.

DATA TYPES

`connection() = handle() | {target_name() (see module ct), connection_type()} | target_`

`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, rx:match 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.

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

Equivalent to cmd(Connection, Cmd, DefaultTimeout) [page 74].

```
cmd(Connection, Cmd, Timeout) -> {ok, Data} | {error, Reason}
```

Types:

- Connection = connection() (see module ct_telnet)
- Cmd = string()
- Timeout = integer()
- Data = [string()]

Send a command via telnet and wait for prompt.

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

Equivalent to cmdf(Connection, CmdFormat, Args, DefaultTimeout) [page 74].

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

Types:

- Connection = connection() (see module ct_telnet)
- CmdFormat = string()
- Args = list()
- Timeout = integer()
- Data = [string()]

Send a telnet command and wait for prompt (uses a format string and list of arguments to build the command).

```
cont_log() -> term()
```

```
end_log() -> term()
```

```
expect(Connection, Patterns) -> term()
```

Equivalent to expect(Connections, Patterns, []) [page 75].

```
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) [page 76].

`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 a node.

`open(Name, ConnType, TargetMod) -> {ok, Handle} | {error, Reason}`

Equivalent to open(Name, ConnType, TargetMod, []) [page 76].

`open(Name, ConnType, TargetMod, Extra) -> {ok, Handle} | {error, Reason}`

Types:

- Name = target_name() (see module ct)
- ConnType = connection_type()
- TargetMod = atom()
- Extra = term()
- Handle = handle()

Open a telnet connection to the specified target.

The target must exist in a config file, and `Name` is the name allocated to the target - either with `ct:require/2` or with a `require` statement in the test suite default data or the test case info function.

`TargetMod` is a module which exports the functions `connect(Ip,Port,Extra)` and `get_prompt_regexp()` for the given `TargetType`.

`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 [page 78]

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},
       {username,UserName},
       {password,Password}]}.
```

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]}).
```

Note that the `{port,PortNum}` tuple is optional and if omitted, default telnet port 23 will be used.

See also

ct [page 50], ct_telnet [page 73]

Index of Modules and Functions

Modules are typed in *this way*.
Functions are typed in `this way`.