Test Server

Copyright © 2002-2013 Ericsson AB. All Rights Reserved.
Test Server 3.6.1
March 19, 2013

Copyright © 2002-2013 Ericsson AB. All Rights Reserved.
The contents of this file are subject to the Erlang Public License, Version 1.1, (the "License"); you may not use this file except in compliance with the License. You should have received a copy of the Erlang Public License along with this software. If not, it can be retrieved online at http://www.erlang.org/. Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License. Ericsson AB. All Rights Reserved..

March 19, 2013

1 Test Server User's Guide

Test Server is a portable test server for automated application testing. The server can run test suites on local or remote targets and log progress and results to HTML pages. The main purpose of Test Server is to act as engine inside customized test tools. A callback interface for such framework applications is provided.

1.1 Test Server Basics

1.1.1 Introduction

Test Server is a portable test tool for automated testing of Erlang programs and OTP applications. It provides an interface for running test programs directly with Test Server as well as an interface for integrating Test Server with a framework application. The latter makes it possible to use Test Server as the engine of a higher level test tool application.

It is strongly recommended that Test Server be used from inside a framework application, rather than interfaced directly for running test programs. Test Server can be pretty difficult to use since it's a very general and quite extensive and complex application. Furthermore, the `test_server_ctrl` functions are not meant to be used from within the actual test programs. The framework should handle communication with Test Server and deal with the more complex aspects of this interaction automatically so that a higher level interface may be provided for the tester. For test tool usage to be productive, a simpler, more intuitive and (if required) more specific interface is required than what Test Server can provide.

OTP delivers a general purpose framework for Test Server, called *Common Test*. This application is a tool well suited for automated black box testing of target systems of *any kind* (not necessarily implemented in Erlang). Common Test is also a very useful tool for white box testing of Erlang programs and OTP applications. Unless a more specific functionality and/or user interface is required (in which case you might need to implement your own framework), Common Test should do the job for you. Please read the Common Test User's Guide and reference manual for more information.

Under normal circumstances, knowledge about the Test Server application is not required for using the Common Test framework. However, if you want to use Test Server without a framework, or learn how to integrate it with your own framework, please read on...

1.1.2 Getting started

Testing when using Test Server is done by running test suites. A test suite is a number of test cases, where each test case tests one or more things. The test case is the smallest unit that the test server deals with. One or more test cases are grouped together into one ordinary Erlang module, which is called a test suite. Several test suite modules can be grouped together in special test specification files representing whole application and/or system test "jobs".

The test suite Erlang module must follow a certain interface, which is specified by Test Server. See the section on writing test suites for details about this.

Each test case is considered a success if it returns to the caller, no matter what the returned value is. An exception to this is the return value `{skip, Reason}` which indicates that the test case is skipped. A failure is specified as a crash, no matter what the crash reason is.

As a test suite runs, all information (including output to stdout) is recorded in several different log files. A minimum of information is displayed to the user console. This only include 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 each test run. Every test case gets one row in a table presenting total time, whether the case was successful or not, if it was skipped, and possibly also a comment. The HTML file has links to each test case's logfile, which may be viewed from e.g. Netscape or any other HTML capable browser.

The Test Server consists of three parts:

- The part that executes the test suites on target and provides support for the test suite author is called `test_server`. This is described in the chapter about writing test cases in this user's guide, and in the reference manual for the `test_server` module.
- The controlling part, which provides the low level operator interface, starts and stops the target node (if remote target) and slave nodes and writes log files, is called `test_server_ctrl`. The Test Server Controller should not be used directly when running tests. Instead a framework built on top of it should be used. More information about how to write your own framework can be found in this user's guide and in the reference manual for the `test_server_ctrl` module.

1.1.3 Definition of terms

conf(iguration) case
: This is a group of test cases which need some specific configuration. A conf case contains an initiation function which sets up a specific configuration, one or more test cases using this configuration, and a cleanup function which restores the configuration. A conf case is specified in a test specification either like this:`{conf,InitFunc,ListOfCases,CleanupFunc}`, or this: `{conf,Properties,InitFunc,ListOfCases,CleanupFunc}`

datadir
: Data directory for a test suite. This directory contains any files used by the test suite, e.g. additional erlang modules, c code or data files. If the data directory contains code which must be compiled before the test suite is run, it should also contain a makefile source called Makefile.src defining how to compile.

documentation clause
: One of the function clauses in a test case. This clause shall return a list of strings describing what the test case tests.

execution clause
: One of the function clauses in a test case. This clause implements the actual test case, i.e. calls the functions that shall be tested and checks results. The clause shall crash if it fails.

major log file
: This is the test suites log file.

Makefile.src
: This file is used by the test server framework to generate a makefile for a datadir. It contains some special characters which are replaced according to the platform currently tested.

minor log file
: This is a separate log file for each test case.

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

skip case
: A test case which shall be skipped.

specification clause
: One of the function clauses in a test case. This clause shall return an empty list, a test specification or `{skip,Reason}`. If an empty list is returned, it means that the test case shall be executed, and so it must also have an execution clause. Note that the specification clause is always executed on the controller node, i.e. not on the target node.

test case
: A single test included in a test suite. Typically it tests one function in a module or application. A test case is implemented as a function in a test suite module. The function can have three clauses, the documentation-, specification- and execution clause.

test specification
: A specification of which test suites and test cases to run. There can be test specifications on three different levels in a test. The top level is a test specification file which roughly specifies what to test for a whole application. Then there is a test specification for each test suite returned from the `all(suite)` function in the suite. And there can also be a test specification returned from the specification clause of a test case.

test specification file
: This is a text file containing the test specification for an application. The file has the extension ".spec" or ".spec.Platform", where Platform is e.g. "vxworks".

test suite
: An erlang module containing a collection of test cases for a specific application or module.

topcase
: The first "command" in a test specification file. This command contains the test specification, like this: `{topcase,TestSpecification}`

1.2 Test Structure and Test Specifications

1.2.1 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.

1.2.2 Test specifications

A test specification is a specification of which test suites and test cases to run and which to skip. A test specification can also group several test cases into conf cases with init and cleanup functions (see section about configuration cases below). In a test there can be test specifications on three different levels:

The top level is a test specification file which roughly specifies what to test for a whole application. The test specification in such a file is encapsulated in a topcase command.

Then there is a test specification for each test suite, specifying which test cases to run within the suite. The test specification for a test suite is returned from the `all(suite)` function in the test suite module.

And finally there can be a test specification per test case, specifying sub test cases to run. The test specification for a test case is returned from the specification clause of the test case.

When a test starts, the total test specification is built in a tree fashion, starting from the top level test specification.

The following are the valid elements of a test specification. The specification can be one of these elements or a list with any combination of the elements:

`{Mod, Case}`
: This specifies the test case Mod:Case/1

`{dir, Dir}`
: This specifies all modules `*_SUITE` in the directory `Dir`

`{dir, Dir, Pattern}`
: This specifies all modules `Pattern*` in the directory `Dir`

`{conf, Init, TestSpec, Fin}`
: This is a configuration case. In a test specification file, `Init` and `Fin` must be `{Mod,Func}`. Inside a module they can also be just `Func`. See the section named Configuration Cases below for more information about this.

`{conf, Properties, Init, TestSpec, Fin}`
: This is a configuration case as explained above, but which also takes a list of execution properties for its group of test cases and nested sub-groups.

`{make, Init, TestSpec, Fin}`
: This is a special version of a conf case which is only used by the test server framework `ts`. `Init` and `Fin` are make and unmake functions for a data directory. `TestSpec` is the test specification for the test suite owning the data directory in question. If the make function fails, all tests in the test suite are skipped. The difference between this "make case" and a normal conf case is that for the make case, `Init` and `Fin` are given with arguments (`{Mod,Func,Args}`), and that they are executed on the controller node (i.e. not on target).

`Case`
: This can only be used inside a module, i.e. not a test specification file. It specifies the test case `CurrentModule:Case`.

1.2.3 Test Specification Files

A test specification file is a text file containing the top level test specification (a topcase command), and possibly one or more additional commands. A "command" in a test specification file means a key-value tuple ended by a dot-newline sequence.

The following commands are valid:

`{topcase, TestSpec}`
: This command is mandatory in all test specification files. `TestSpec` is the top level test specification of a test.

`{skip, {Mod, Comment}}`
: This specifies that all cases in the module `Mod` shall be skipped. `Comment` is a string.

`{skip, {Mod, Case, Comment}}`
: This specifies that the case `Mod:Case` shall be skipped.

`{skip, {Mod, CaseList, Comment}}`
: This specifies that all cases `Mod:Case`, where `Case` is in `CaseList`, shall be skipped.

`{nodes, Nodes}`
: `Nodes` is a list of nodenames available to the test suite. It will be added to the `Config` argument to all test cases. `Nodes` is a list of atoms.

`{require_nodenames, Num}`
: Specifies how many nodenames the test suite will need. Theese will be automatically generated and inserted into the `Config` argument to all test cases. `Num` is an integer.

`{hosts, Hosts}`
: This is a list of available hosts on which to start slave nodes. It is used when the `{remote, true}` option is given to the `test_server:start_node/3` function. Also, if `{require_nodenames, Num}` is contained in a test specification file, the generated nodenames will be spread over all hosts given in this `Hosts` list. The hostnames are atoms or strings.

`{diskless, true}`
: Adds `{diskless, true}` to the `Config` argument to all test cases. This is kept for backwards compatibility and should not be used. Use a configuration case instead.

`{ipv6_hosts, Hosts}`
: Adds `{ipv6_hosts, Hosts}` to the `Config` argument to all test cases.

All test specification files shall have the extension ".spec". If special test specification files are needed for Windows or VxWorks platforms, additional files with the extension ".spec.win" and ".spec.vxworks" shall be used. This is useful e.g. if some test cases shall be skipped on these platforms.

Some examples for test specification files can be found in the Examples section of this user's guide.

1.2.4 Configuration cases

If a group of test cases need the same initialization, a so called *configuration* or *conf* case can be used. A conf case consists of an initialization function, the group of test cases needing this initialization and a cleanup or finalization function.

If the init function in a conf case fails or returns `{skip,Comment}`, the rest of the test cases in the conf case (including the cleanup function) are skipped. If the init function succeeds, the cleanup function will always be called, even if some of the test cases in between failed.

Both the init function and the cleanup function in a conf case get the `Config` parameter as only argument. This parameter can be modified or returned as is. Whatever is returned by the init function is given as `Config` parameter to the rest of the test cases in the conf case, including the cleanup function.

If the `Config` parameter is changed by the init function, it must be restored by the cleanup function. Whatever is returned by the cleanup function will be given to the next test case called.

The optional `Properties` list can be used to specify execution properties for the test cases and possibly nested sub-groups of the configuration case. The available properties are:

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

If the `parallel` property is specified, Test Server will execute all test cases in the group in parallel. If `sequence` is specified, the cases will be executed in a sequence, meaning if one case fails, all following cases will be skipped. If `shuffle` is specified, the cases in the group will be executed in random order. The `repeat` property orders Test Server to repeat execution of the cases in the group a given number of times, or until any, or all, cases fail or succeed.

Properties may be combined so that e.g. if `shuffle`, `repeat_until_any_fail` and `sequence` are all specified, the test cases in the group will be executed repeatedly and in random order until a test case fails, when execution is immediately stopped and the rest of the cases skipped.

The properties for a conf case is always printed on the top of the HTML log for the group's init function. Also, the total execution time for a conf case can be found at the bottom of the log for the group's end function.

Configuration cases may be nested so that sets of grouped cases can be configured with the same init- and end functions.

1.2.5 The parallel property and nested configuration cases

If a conf case has a parallel property, its test cases will be spawned simultaneously and get executed in parallel. A test case is not allowed to execute in parallel with the end function however, which means that the time it takes to execute a set of parallel cases is equal to the execution time of the slowest test case in the group. A negative side effect of running test cases in parallel is that the HTML summary pages are not updated with links to the individual test case logs until the end function for the conf case has finished.

A conf case nested under a parallel conf case will start executing in parallel with previous (parallel) test cases (no matter what properties the nested conf case has). Since, however, test cases are never executed in parallel with the init- or the end function of the same conf case, it's only after a nested group of cases has finished that any remaining parallel cases in the previous conf case get spawned.

1.2.6 Repeated execution of test cases

A conf case may be repeated a certain number of times (specified by an integer) or indefinitely (specified by `forever`). The repetition may also be stopped prematurely if any or all cases fail or succeed, i.e. if the property `repeat_until_any_fail`, `repeat_until_any_ok`, `repeat_until_all_fail`, or `repeat_until_all_ok` is used. If the basic `repeat` property is used, status of test cases is irrelevant for the repeat operation.

It is possible to return the status of a conf case (ok or failed), to affect the execution of the conf case on the level above. This is accomplished by, in the end function, looking up the value of `tc_group_properties` in the `Config` list and checking the result of the finished test cases. If status `failed` should be returned from the conf case as a result, the end function should return the value `{return_group_result,failed}`. The status of a nested conf case is taken into account by Test Server when deciding if execution should be repeated or not (unless the basic `repeat` property is used).

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

Here's an example of how to return the status from a conf case:

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

It is also possible in the end function to check the status of a nested conf case (maybe to determine what status the current conf case should return). This is as simple as illustrated in the example above, only the name of the end function of the nested conf case is stored in a tuple `{group_result,EndFunc}`, which can be searched for in the status lists. Example:

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

> **Note:**
>
> When a conf case is repeated, the init- and end functions are also always called with each repetition.

1.2.7 Shuffled test case order

The order that test cases in a conf case are executed, is under normal circumstances the same as the order defined in the test specification. With the `shuffle` property set, however, Test Server will instead execute the test cases in random order.

The user may provide a seed value (a tuple of three integers) with the shuffle property: `{shuffle,Seed}`. This way, the same shuffling order can be created every time the conf case is executed. If no seed value is given, Test Server creates a "random" seed for the shuffling operation (using the return value of `erlang:now()`). The seed value is always printed to the log file of the init function so that it can be used to recreate the same execution order in subsequent test runs.

> **Note:**
>
> If execution of a conf case with shuffled test cases is repeated, the seed will not be reset in between turns.

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

1.2.8 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 a test case should be skipped:

- Using the `{skip,What}` command in a test specification file
- Returning `{skip,Reason}` from the `init_per_testcase/2` function
- Returning `{skip,Reason}` from the specification clause of the test case
- 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. For more information about the different clauses in a test case, see the chapter about writing test cases.

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

1.3 Writing Test Suites

1.3.1 Support for test suite authors

The `test_server` module provides some useful functions to support the test suite author. This includes:

- Starting and stopping slave or peer nodes
- Capturing and checking stdout output
- Retrieving and flushing process message queue
- Watchdog timers
- Checking that a function crashes
- Checking that a function succeeds at least m out of n times
- Checking .app files

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

1.3.2 Test suites

A test suite is an ordinary Erlang module that contains test cases. It's recommended that the module has a name on the form *_SUITE.erl. Otherwise, the directory function will not find the modules (by default).

For some of the test server support, the test server include file `test_server.hrl` must be included. Never include it with the full path, for portability reasons. Use the compiler include directive instead.

The special function `all(suite)` in each module is called to get the test specification for that module. The function typically returns a list of test cases in that module, but any test specification could be returned. Please see the chapter about test specifications for details about this.

1.3.3 Init per test case

In each test suite module, the functions `init_per_testcase/2` and `end_per_testcase/2` must be implemented.

`init_per_testcase` is called before each test case in the test suite, giving a (limited) possibility for initialization.

`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 initialization and cleanup for each test cases.

The second argument is a list of tuples called `Config`. The first element in a `Config` tuple should be an atom - a key value to be used for searching. `init_per_testcase/2` may modify the `Config` 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.

1.3.4 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 tests is up to the author, but here are some things to keep in mind.

Very small test cases often leads to more code, since initialization has to be duplicated. Larger code, especially with a lot of duplication, increases maintenance and reduces readability.

Larger test cases make it harder to tell what went wrong if it fails, and force us to skip larger portions of test code if a specific part fails. These effects are accentuated when running on multiple platforms because test cases often have to be skipped.

A test case generally consists of three parts, the documentation part, the specification part and the execution part. These are implemented as three clauses of the same function.

The documentation clause matches the argument '`doc`' and returns a list for strings describing what the test case tests.

The specification clause matches the argument '`suite`' and returns the test specification for this particular test case. If the test specification is an empty list, this indicates that the test case is a leaf test case, i.e. one to be executed.

Note that the specification clause of a test case is executed on the test server controller host. This means that if target is remote, the specification clause is probably executed on a different platform than the one tested.

The execution clause implements the actual test case. It takes one argument, `Config`, which contain configuration information like `data_dir` and `priv_dir`. See *Data and Private Directories* for more information about these.

The `Config` variable can also contain the `nodenames` key, if requested by the `require_nodenames` command in the test suite specification file. All `Config` items should be extracted using the `?config` macro. This is to ensure

future compatibility if the `Config` format changes. See the reference manual for `test_server` for details about this macro.

If the execution clause crashes or exits, it is considered a failure. If it returns `{skip,Reason}`, the test case is considered skipped. If it returns `{comment,String}`, the string will be added in the 'Comment' field on the HTML result page. If the execution clause returns anything else, it is considered a success, unless it is `{'EXIT',Reason}` or `{'EXIT',Pid,Reason}` which can't be distinguished from a crash, and thus will be considered a failure.

1.3.5 Data and Private Directories

The data directory (`data_dir`) is the directory where the test module has its own files needed for the testing. A compiler test case may have source files to feed into the compiler, a release upgrade test case may have some old and new release of something. A graphics test case may have some icons and a test case doing a lot of math with bignums might store the correct answers there. The name of the `data_dir` is the the name of the test suite and then "_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.

Warning: Do not depend on current directory to be writable, or to point to anything in particular. All scratch files are to be written in the `priv_dir`, and all data files found in `data_dir`. If the current directory has to be something specific, it must be set with `file:set_cwd/1`.

1.3.6 Execution environment

Each time a test case is about to be executed, a new process is created with `spawn_link`. This is so that the test case will have no dependencies to earlier tests, with respect to process flags, process links, messages in the queue, other processes having registered the process, etc. As little as possible is done to change the initial context of the process (what is created by plain spawn). Here is a list of differences:

- It has a link to the test server. If this link is removed, the test server will not know when the test case is finished, just wait infinitely.
- It often holds a few items in the process dictionary, all with names starting with '`test_server_`'. This is to keep track of if/where a test case fails.
- There is a top-level catch. All of the test case code is catched, so that the location of a crash can be reported back to the test server. If the test case process is killed by another process (thus the catch code is never executed) the test server is not able to tell where the test case was executing.
- It has a special group leader implemented by the test server. This way the test server is able to capture the io that the test case provokes. This is also used by some of the test server support functions.

There is no time limit for a test case, unless the test case itself imposes such a limit, by calling `test_server:timetrap/1` for example. The call can be made in each test case, or in the `init_per_testcase/2` function. Make sure to call the corresponding `test_server:timetrap_cancel/1` function as well, e.g in the `end_per_testcase/2` function, or else the test cases will always fail.

1.4 Running Test Suites

1.4.1 Using the test server controller

The test server controller provides a low level interface to all the Test Server functionality. It is possible to use this interface directly, but it is recommended to use a framework such as *Common Test* instead. If no existing framework suits your needs, you could of course build your own on top of the test server controller. Some information about how to do this can be found in the section named "Writing you own test server framework" in the Test Server User's Guide.

For information about using the controller directly, please see all available functions in the reference manual for `test_server_ctrl`.

1.5 Write you own test server framework

1.5.1 Introduction

The test server controller can be interfaced from the operating system or from within Erlang. The nature of your new framework will decide which interface to use. If you want your framework to start a new node for each test, the operating system interface is very convenient. If your node is already started, going from within Erlang might be a more flexible solution.

The two methods are described below.

1.5.2 Interfacing the test server controller from Erlang

Using the test server from Erlang means that you have to start the test server and then add test jobs. Use `test_server_ctrl:start/0` to start a local target or `test_server_ctrl:start/1` to start a remote target. The test server is stopped by `test_server_ctrl:stop/0`.

The argument to `test_server_ctrl:start/1` is the name of a parameter file. The parameter file specifies what type of target to start and where to start it, as well as some additional parameters needed for different target types. See the reference manual for a detailed description of all valid parameters.

Adding test jobs

There are many commands available for adding test cases to the test server's job queue:

- Single test case
  `test_server_ctrl:add_case/2/3`
- Multiple test cases from same suite
  `test_server_ctrl:add_cases/2/3`
- Test suite module or modules
  `test_server_ctrl:add_module/1/2`
- Some or all test suite modules in a directory
  `test_server_ctrl:add_dir/2/3`
- Test cases specified in a test specification file
  `test_server_ctrl:add_spec/1`

All test suites are given a unique name, which is usually given when the test suite is added to the job queue. In some cases, a default name is used, as in the case when a module is added without a specified name. The test job name is used to store logfiles, which are stored in the `name.logs' directory under the current directory.

See the reference manual for details about the functions for adding test jobs.

1.5.3 Interfacing the test server controller from the operating system.

The function `run_test/1` is your interface in the test server controller if you wish to use it from the operating system. You simply start an erlang shell and invoke this function with the `-s` option. `run_test/1` starts the test server, runs the test specified by the command line and stops the test server. The argument to `run_test/1` is a list of command line flags, typically `['KEY1', Value1, 'KEY2', Value2, ...]`. The valid command line flags are listed in the reference manual for `test_server_ctrl`.

A typical command line may look like this

```
erl -noshell -s test_server_ctrl run_test KEY1 Value1 KEY2 Value2 ... -s erlang
halt
```

Or make an alias (this is for unix/tcsh)

```
alias erl_test 'erl -noshell -s test_server_ctrl run_test \!* -s erlang halt'
```

And then use it like this

```
erl_test KEY1 Value1 KEY2 Value2 ...
```

An Example

An example of starting a test run from the command line

```
erl -name test_srv -noshell -rsh /home/super/otp/bin/ctrsh
-pa /clearcase/otp/erts/lib/kernel/test
-boot start_sasl -sasl errlog_type error
-s test_server_ctrl run_test SPEC kernel.spec -s erlang halt
```

1.5.4 Framework callback functions

By defining the environment variable `TEST_SERVER_FRAMEWORK` to a module name, the framework callback functions can be used. The framework callback functions are called by the test server in order let the framework interact with the execution of the tests and to keep the framework upto date with information about the test progress.

The framework callback functions are described in the reference manual for `test_server_ctrl`.

Note that this topic is in an early stage of development, and changes might occur.

1.5.5 Other concerns

Some things to think about when writing you own test server framework:

- `emulator version` - Make sure that the intended version of the emulator is started.
- `operating system path` - If test cases use port programs, make sure the paths are correct.
- `recompilation` - Make sure all test suites are fresh compiled.
- `test_server.hrl` - Make sure the `test_server.hrl` file is in the include path when compiling test suites.
- `running applications` - Some test suites require some applications to be running (e.g. sasl). Make sure they are started.

1.6 Examples

1.6.1 Test suite

```
-module(my_SUITE).

-export([all/1,
         not_started/1, not_started_func1/1, not_started_func2/1,
         start/1, stop/1,
         func1/1, func2/1
        ]).

-export([init_per_testcase/2, end_per_testcase/2]).

-include("test_server.hrl").

-define(default_timeout, ?t:minutes(1)).

init_per_testcase(_Case, Config) ->
    ?line Dog=?t:timetrap(?default_timeout),
```

```
    [{watchdog, Dog}|Config].
end_per_testcase(_Case, Config) ->
    Dog=?config(watchdog, Config),
    ?t:timetrap_cancel(Dog),
    ok.

all(suite) ->
    %% Test specification on test suite level
    [not_started,
     {conf, start, [func1, func2], stop}].

not_started(suite) ->
    %% Test specification on test case level
    [not_started_func1, not_started_func2];
not_started(doc) ->
    ["Testing all functions when application is not started"].
%% No execution clause unless the specification clause returns [].


not_started_func1(suite) ->
    [];
not_started_func1(doc) ->
    ["Testing function 1 when application is not started"].
not_started_func1(Config) when list(Config) ->
    ?line {error, not_started} = myapp:func1(dummy_ref,1),
    ?line {error, not_started} = myapp:func1(dummy_ref,2),
    ok.

not_started_func2(suite) ->
    [];
not_started_func2(doc) ->
    ["Testing function 2 when application is not started"].
not_started_func2(Config) when list(Config) ->
    ?line {error, not_started} = myapp:func2(dummy_ref,1),
    ?line {error, not_started} = myapp:func2(dummy_ref,2),
    ok.


%% No specification clause needed for an init function in a conf case!!!
start(doc) ->
    ["Testing start of my application."];
start(Config) when list(Config) ->
    ?line Ref = myapp:start(),
    case erlang:whereis(my_main_process) of
      Pid when pid(Pid) ->
         [{myapp_ref,Ref}|Config];
      undefined ->
         %% Since this is the init function in a conf case, the rest of the
         %% cases in the conf case will be skipped if this case fails.
         ?t:fail("my_main_process did not start")
    end.

func1(suite) ->
    [];
func1(doc) ->
    ["Test that func1 returns ok when argument is 1 and error if argument is 2"];
func1(Config) when list(Config) ->
    ?line Ref = ?config(myapp_ref,Config),
    ?line ok = myapp:func1(Ref,1),
    ?line error = myapp:func1(Ref,2),
    ok.

func2(suite) ->
    [];
func2(doc) ->
```

```
    ["Test that func1 returns ok when argument is 3 and error if argument is 4"];
func2(Config) when list(Config) ->
    ?line Ref = ?config(myapp_ref,Config),
    ?line ok = myapp:func2(Ref,3),
    ?line error = myapp:func2(Ref,4),
    ok.

%% No specification clause needed for a cleanup function in a conf case!!!
stop(doc) ->
    ["Testing termination of my application"];
stop(Config) when list(Config) ->
    ?line Ref = ?config(myapp_ref,Config),
    ?line ok = myapp:stop(Ref),
    case erlang:whereis(my_main_process) of
      undefined ->
         lists:keydelete(myapp_ref,1,Config);
      Pid when pid(Pid) ->
         ?t:fail("my_main_process did not stop")
    end.
```

1.6.2 Test specification file

myapp.spec:

```
{topcase, {dir, "../myapp_test"}}. % Test specification on top level
```

myapp.spec.vxworks:

```
{topcase, {dir, "../myapp_test"}}. % Test specification on top level
{skip,{my_SUITE,func2,"Not applicable on VxWorks"}}.
```

2 Reference Manual

Test Server is a portable test server for automated application testing. The server can run test suites on local or remote targets and log progress and results to HTML pages. The main purpose of Test Server is to act as engine inside customized test tools. A callback interface for such framework applications is provided.

test_server

Application

Test Server is a portable test server for automated application testing. The server can run test suites on local or remote targets and log progress and results to HTML pages. The main purpose of Test Server is to act as engine inside customized test tools. A callback interface for such framework applications is provided.

In brief the test server supports:

- Running multiple, concurrent test suites
- Running tests on remote and even diskless targets
- Test suites may contain other test suites, in a tree fashion
- 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 for test suite authors, e.g. start/stop slave nodes
- Call trace on target and slave nodes

For information about how to write test cases and test suites, please see the Test Server User's Guide and the reference manual for the `test_server` module.

Common Test is an existing test tool application based on the OTP Test Server. Please read the Common Test User's Guide for more information.

Configuration

There are currently no configuration parameters available for this application.

SEE ALSO

test_server_ctrl

Erlang module

The `test_server_ctrl` module provides a low level interface to the Test Server. This interface is normally not used directly by the tester, but through a framework built on top of `test_server_ctrl`.

Common Test is such a framework, well suited for automated black box testing of target systems of any kind (not necessarily implemented in Erlang). Common Test is also a very useful tool for white box testing Erlang programs and OTP applications. Please see the Common Test User's Guide and reference manual for more information.

If you want to write your own framework, some more information can be found in the chapter "Writing your own test server framework" in the Test Server User's Guide. Details about the interface provided by `test_server_ctrl` follows below.

Exports

```
start() -> Result
start(ParameterFile) -> Result
```

Types:

`Result = ok | {error, {already_started, pid()}`
`ParameterFile = atom() | string()`

This function starts the test server. If the parameter file is given, it indicates that the target is remote. In that case the target node is started and a socket connection is established between the controller and the target node.

The parameter file is a text file containing key-value tuples. Each tuple must be followed by a dot-newline sequence. The following key-value tuples are allowed:

`{type,PlatformType}`
: This is an atom indicating the target platform type, currently supported: `PlatformType = vxworks`
Mandatory

`{target,TargetHost}`
: This is the name of the target host, can be atom or string.
Mandatory

`{slavetargets,SlaveTargets}`
: This is a list of available hosts where slave nodes can be started. The hostnames are given as atoms or strings.
Optional, default `SlaveTargets = []`

`{longnames,Bool}`
: This indicates if longnames shall be used, i.e. if the `-name` option should be used for the target node instead of `-sname`
Optional, default `Bool = false`

`{master, {MasterHost, MasterCookie}}`
: If target is remote and the target node is started as a slave node, this option indicates which master and cookie to use. The given master will also be used as master for slave nodes started with `test_server:start_node/3`. It is expected that the `erl_boot_server` is started on the master node before the `test_server_ctrl:start/1` function is called.
Optional, if not given the test server controller node is used as master and the `erl_boot_server` is automatically started.

```
stop() -> ok
```

This stops the test server (both controller and target) and all its activity. The running test suite (if any) will be halted.

```
add_dir(Name, Dir) -> ok
add_dir(Name, Dir, Pattern) -> ok
add_dir(Name, [Dir|Dirs]) -> ok
add_dir(Name, [Dir|Dirs], Pattern) -> ok
```

Types:

`Name = term()`

The jobname for this directory.

`Dir = term()`

The directory to scan for test suites.

`Dirs = [term()]`

List of directories to scan for test suites.

`Pattern = term()`

Suite match pattern. Directories will be scanned for Pattern_SUITE.erl files.

Puts a collection of suites matching (*_SUITE) in given directories into the job queue. `Name` is an arbitrary name for the job, it can be any erlang term. If `Pattern` is given, only modules matching `Pattern*` will be added.

```
add_module(Mod) -> ok
add_module(Name, [Mod|Mods]) -> ok
```

Types:

`Mod = atom()`

`Mods = [atom()]`

The name(s) of the module(s) to add.

`Name = term()`

Name for the job.

This function adds a module or a list of modules, to the test servers job queue. `Name` may be any Erlang term. When `Name` is not given, the job gets the name of the module.

```
add_case(Mod, Case) -> ok
```

Types:

`Mod = atom()`

Name of the module the test case is in.

`Case = atom()`

Function name of the test case to add.

This function will add one test case to the job queue. The job will be given the module's name.

```
add_case(Name, Mod, Case) -> ok
```

Types:

`Name = string()`

Name to use for the test job.

Equivalent to `add_case/2`, but the test job will get the specified name.

```
add_cases(Mod, Cases) -> ok
```

Types:

Mod = atom()

Name of the module the test case is in.

Cases = [Case]

Case = atom()

Function names of the test cases to add.

This function will add one or more test cases to the job queue. The job will be given the module's name.

```
add_cases(Name, Mod, Cases) -> ok
```

Types:

Name = string()

Name to use for the test job.

Equivalent to `add_cases/2`, but the test job will get the specified name.

```
add_spec(TestSpecFile) -> ok | {error, nofile}
```

Types:

TestSpecFile = string()

Name of the test specification file

This function will add the content of the given test specification file to the job queue. The job will be given the name of the test specification file, e.g. if the file is called `test.spec`, the job will be called `test`.

See the reference manual for the test server application for details about the test specification file.

```
add_dir_with_skip(Name, [Dir|Dirs], Skip) -> ok
add_dir_with_skip(Name, [Dir|Dirs], Pattern, Skip) -> ok
add_module_with_skip(Mod, Skip) -> ok
add_module_with_skip(Name, [Mod|Mods], Skip) -> ok
add_case_with_skip(Mod, Case, Skip) -> ok
add_case_with_skip(Name, Mod, Case, Skip) -> ok
add_cases_with_skip(Mod, Cases, Skip) -> ok
add_cases_with_skip(Name, Mod, Cases, Skip) -> ok
```

Types:

Skip = [SkipItem]

List of items to be skipped from the test.

SkipItem = {Mod,Comment} | {Mod,Case,Comment} | {Mod,Cases,Comment}

Mod = atom()

Test suite name.

Comment = string()

Reason why suite or case is being skipped.

Cases = [Case]

Case = atom()

Name of test case function.

These functions add test jobs just like the add_dir, add_module, add_case and add_cases functions above, but carry an additional argument, Skip. Skip is a list of items that should be skipped in the current test run. Test job items that occur in the Skip list will be logged as SKIPPED with the associated Comment.

`add_tests_with_skip(Name, Tests, Skip) -> ok`

Types:

`Name = term()`

The jobname for this directory.

`Tests = [TestItem]`

List of jobs to add to the run queue.

`TestItem = {Dir,all,all} | {Dir,Mods,all} | {Dir,Mod,Cases}`

`Dir = term()`

The directory to scan for test suites.

`Mods = [Mod]`

`Mod = atom()`

Test suite name.

`Cases = [Case]`

`Case = atom()`

Name of test case function.

`Skip = [SkipItem]`

List of items to be skipped from the test.

`SkipItem = {Mod,Comment} | {Mod,Case,Comment} | {Mod,Cases,Comment}`

`Comment = string()`

Reason why suite or case is being skipped.

This function adds various test jobs to the test_server_ctrl job queue. These jobs can be of different type (all or specific suites in one directory, all or specific cases in one suite, etc). It is also possible to get particular items skipped by passing them along in the Skip list (see the add_*_with_skip functions above).

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

Types:

`Reason = term()`

The reason for stopping the test case, which will be printed in the log.

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

`set_levels(Console, Major, Minor) -> ok`

Types:

`Console = integer()`

Level for I/O to be sent to console.

`Major = integer()`

Level for I/O to be sent to the major logfile.

`Minor = integer()`

Level for I/O to be sent to the minor logfile.

Determines where I/O from test suites/test server will go. All text output from test suites and the test server is tagged with a priority value which ranges from 0 to 100, 100 being the most detailed. (see the section about log files in the user's guide). Output from the test cases (using `io:format/2`) has a detail level of 50. Depending on the levels

set by this function, this I/O may be sent to the console, the major log file (for the whole test suite) or to the minor logfile (separate for each test case).

All output with detail level:

- Less than or equal to `Console` is displayed on the screen (default 1)
- Less than or equal to `Major` is logged in the major log file (default 19)
- Greater than or equal to `Minor` is logged in the minor log files (default 10)

To view the currently set thresholds, use the `get_levels/0` function.

```
get_levels() -> {Console, Major, Minor}
```

Returns the current levels. See `set_levels/3` for types.

```
jobs() -> JobQueue
```

Types:

`JobQueue = [{list(), pid()}]`

This function will return all the jobs currently in the job queue.

```
multiply_timetraps(N) -> ok
```

Types:

`N = integer() | infinity`

This function should be called before a test is started which requires extended timetraps, e.g. if extensive tracing is used. All timetraps started after this call will be multiplied by `N`.

```
scale_timetraps(Bool) -> ok
```

Types:

`Bool = true | false`

This function should be called before a test is started. The parameter specifies if test_server should attempt to automatically scale the timetrap value in order to compensate for delays caused by e.g. the cover tool.

```
get_timetrap_parameters() -> {N,Bool}
```

Types:

`N = integer() | infinity`

`Bool = true | false`

This function may be called to read the values set by `multiply_timetraps/1` and `scale_timetraps/1`.

```
cover(Application,Analyse) -> ok
cover(CoverFile,Analyse) -> ok
cover(App,CoverFile,Analyse) -> ok
```

Types:

`Application = atom()`

OTP application to cover compile

`CoverFile = string()`

Name of file listing modules to exclude from or include in cover compilation. The filename must include full path to the file.

Analyse = details | overview

This function informs the test_server controller that next test shall run with code coverage analysis. All timetraps will automatically be multiplied by 10 when cover i run.

`Application` and `CoverFile` indicates what to cover compile. If `Application` is given, the default is that all modules in the `ebin` directory of the application will be cover compiled. The `ebin` directory is found by adding `ebin` to `code:lib_dir(Application)`.

A `CoverFile` can have the following entries:

```
{exclude, all | ExcludeModuleList}.
{include, IncludeModuleList}.
{cross, CrossCoverInfo}.
```

Note that each line must end with a full stop. `ExcludeModuleList` and `IncludeModuleList` are lists of atoms, where each atom is a module name.

`CrossCoverInfo` is used when collecting cover data over multiple tests. Modules listed here are compiled, but they will not be analysed when the test is finished. See *cross_cover_analyse/2* for more information about the cross cover mechanism and the format of `CrossCoverInfo`.

If both an `Application` and a `CoverFile` is given, all modules in the application are cover compiled, except for the modules listed in `ExcludeModuleList`. The modules in `IncludeModuleList` are also cover compiled.

If a `CoverFile` is given, but no `Application`, only the modules in `IncludeModuleList` are cover compiled.

`Analyse` indicates the detail level of the cover analysis. If `Analyse = details`, each cover compiled module will be analysed with `cover:analyse_to_file/1`. If `Analyse = overview` an overview of all cover compiled modules is created, listing the number of covered and not covered lines for each module.

If the test following this call starts any slave or peer nodes with `test_server:start_node/3`, the same cover compiled code will be loaded on all nodes. If the loading fails, e.g. if the node runs an old version of OTP, the node will simply not be a part of the coverage analysis. Note that slave or peer nodes must be stopped with `test_server:stop_node/1` for the node to be part of the coverage analysis, else the test server will not be able to fetch coverage data from the node.

When the test is finished, the coverage analysis is automatically completed, logs are created and the cover compiled modules are unloaded. If another test is to be run with coverage analysis, `test_server_ctrl:cover/2/3` must be called again.

`cross_cover_analyse(Level, Tests) -> ok`

Types:

Level = details | overview

Tests = [{Tag,LogDir}]

Tag = atom()

Test identifier.

LogDir = string()

Log directory for the test identified by `Tag`. This can either be the `run.<timestamp>` directory or the parent directory of this (in which case the latest `run.<timestamp>` directory is chosen.

Analyse cover data collected from multiple tests. The modules analysed are the ones listed in `cross` statements in the cover files. These are modules that are heavily used by other tests than the one where they belong or are explicitly tested. They should then be listed as cross modules in the cover file for the test where they are used but do not belong. Se example below.

This function should be run after all tests are completed, and the result will be stored in a file called `cross_cover.html` in the `run.<timestamp>` directory of the test the modules belong to.

Note that the function can be executed on any node, and it does not require `test_server_ctrl` to be started first.

The `cross` statement in the cover file must be like this:

```
{cross,[{Tag,Modules}]}.
```

where `Tag` is the same as `Tag` in the `Tests` parameter to this function and `Modules` is a list of module names (atoms).

Example:

If the module `m1` belongs to system `s1` but is heavily used also in the tests for another system `s2`, then the cover files for the two systems' tests could be like this:

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

s2.cover:
  {include,[....]}. % modules belonging to system s2
  {cross,[{s1,[m1]}]}.
```

When the tests for both `s1` and `s2` are completed, run

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

and the accumulated cover data for `m1` will be written to `S1LogDir/[run.<timestamp>/]cross_cover.html`.

Note that the `m1` module will also be presented in the normal coverage log for `s1` (due to the include statement in `s1.cover`), but that only includes the coverage achieved by the `s1` test itself.

The Tag in the `cross` statement in the cover file has no other purpose than mapping the list of modules (`[m1]` in the example above) to the correct log directory where it should be included in the `cross_cover.html` file (`S1LogDir` in the example above). I.e. the value of `Tag` has no meaning, it could be `foo` as well as `s1` above, as long as the same `Tag` is used in the cover file and in the call to this function.

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

Types:

`TraceInfoFile = atom() | string()`

Name of a file defining which functions to trace and how

This function starts call trace on target and on slave or peer nodes that are started or will be started by the test suites.

Timetraps are not extended automatically when tracing is used. Use `multiply_timetraps/1` if necessary.

Note that the trace support in the test server is in a very early stage of the implementation, and thus not yet as powerful as one might wish for.

The trace information file specified by the `TraceInfoFile` argument is a text file containing one or more of the following elements:

- `{SetTP,Module,Pattern}.`

- `{SetTP,Module,Function,Pattern}.`
- `{SetTP,Module,Function,Arity,Pattern}.`
- `ClearTP.`
- `{ClearTP,Module}.`
- `{ClearTP,Module,Function}.`
- `{ClearTP,Module,Function,Arity}.`

`SetTP = tp | tpl`
: This is maps to the corresponding functions in the `ttb` module in the `observer` application. `tp` means set trace pattern on global function calls. `tpl` means set trace pattern on local and global function calls.

`ClearTP = ctp | ctpl | ctpg`
: This is maps to the corresponding functions in the `ttb` module in the `observer` application. `ctp` means clear trace pattern (i.e. turn off) on global and local function calls. `ctpl` means clear trace pattern on local function calls only and `ctpg` means clear trace pattern on global function calls only.

`Module = atom()`
: The module to trace

`Function = atom()`
: The name of the function to trace

`Arity = integer()`
: The arity of the function to trace

`Pattern = [] | match_spec()`
: The trace pattern to set for the module or function. For a description of the match_spec() syntax, please turn to the User's guide for the runtime system (erts). The chapter "Match Specification in Erlang" explains the general match specification language.

The trace result will be logged in a (binary) file called `NodeName-test_server` in the current directory of the test server controller node. The log must be formatted using `ttb:format/1/2`.

```
stop_trace() -> ok | {error, not_tracing}
```

This function stops tracing on target, and on slave or peer nodes that are currently running. New slave or peer nodes will no longer be traced after this.

FUNCTIONS INVOKED FROM COMMAND LINE

The following functions are supposed to be invoked from the command line using the `-s` option when starting the erlang node.

Exports

```
run_test(CommandLine) -> ok
```

Types:

`CommandLine = FlagList`

This function is supposed to be invoked from the commandline. It starts the test server, interprets the argument supplied from the commandline, runs the tests specified and when all tests are done, stops the test server and returns to the Erlang prompt.

The `CommandLine` argument is a list of command line flags, typically `['KEY1', Value1, 'KEY2', Value2, ...]`. The valid command line flags are listed below.

Under a UNIX command prompt, this function can be invoked like this:

```
erl -noshell -s test_server_ctrl run_test KEY1 Value1 KEY2 Value2 ... -s erlang
halt
```

Or make an alias (this is for unix/tcsh)

```
alias erl_test 'erl -noshell -s test_server_ctrl run_test \!* -s erlang halt'
```

And then use it like this

```
erl_test KEY1 Value1 KEY2 Value2 ...
```

The valid command line flags are

`DIR dir`
: Adds all test modules in the directory `dir` to the job queue.

`MODULE mod`
: Adds the module `mod` to the job queue.

`CASE mod case`
: Adds the case `case` in module `mod` to the job queue.

`SPEC spec`
: Runs the test specification file `spec`.

`SKIPMOD mod`
: Skips all test cases in the module `mod`

`SKIPCASE mod case`
: Skips the test case `case` in module `mod`.

`NAME name`
: Names the test suite to something else than the default name. This does not apply to `SPEC` which keeps its names.

`PARAMETERS parameterfile`
: Specifies the parameter file to use when starting remote target

`COVER app cover_file analyse`
: Indicates that the test should be run with cover analysis. `app`, `cover_file` and `analyse` corresponds to the parameters to `test_server_ctrl:cover/3`. If no cover file is used, the atom `none` should be given.

`TRACE traceinfofile`
: Specifies a trace information file. When this option is given, call tracing is started on the target node and all slave or peer nodes that are started. The trace information file specifies which modules and functions to trace. See the function `trc/1` above for more information about the syntax of this file.

FRAMEWORK CALLBACK FUNCTIONS

A test server framework can be defined by setting the environment variable `TEST_SERVER_FRAMEWORK` to a module name. This module will then be framework callback module, and it must export the following function:

Exports

```
get_suite(Mod,Func) -> TestCaseList
```

Types:

`Mod = atom()`

Test suite name.

`Func = atom()`

Name of test case.

`TestCaseList = [SubCase]`

List of test cases.

`SubCase = atom()`

Name of a case.

This function is called before a test case is started. The purpose is to retrieve a list of subcases. The default behaviour of this function should be to call `Mod:Func(suite)` and return the result from this call.

```
init_tc(Mod,Func,Args0) -> {ok,Args1} | {skip,ReasonToSkip} |
{auto_skip,ReasonToSkip} | {fail,ReasonToFail}
```

Types:

`Mod = atom()`

Test suite name.

`Func = atom()`

Name of test case or configuration function.

`Args0 = Args1 = [tuple()]`

Normally Args = [Config]

`ReasonToSkip = term()`

Reason to skip the test case or configuration function.

`ReasonToFail = term()`

Reason to fail the test case or configuration function.

This function is called before a test case or configuration function starts. It is called on the process executing the function `Mod:Func`. Typical use of this function can be to alter the input parameters to the test case function (`Args`) or to set properties for the executing process.

By returning `{skip,Reason}`, `Func` gets skipped. `Func` also gets skipped if `{auto_skip,Reason}` is returned, but then gets an auto skipped status (rather than user skipped).

To fail `Func` immediately instead of executing it, return `{fail,ReasonToFail}`.

```
end_tc(Mod,Func,Status) -> ok | {fail,ReasonToFail}
```

Types:

`Mod = atom()`

Test suite name.

`Func = atom()`

Name of test case or configuration function.

`Status = {Result,Args} | {TCPid,Result,Args}`

The status of the test case or configuration function.

`ReasonToFail = term()`

Reason to fail the test case or configuration function.

`Result = ok | Skip | Fail`

The final result of the test case or configuration function.

`TCPid = pid()`

Pid of the process executing Func

`Skip = {skip,SkipReason}`

`SkipReason = term() | {failed,{Mod,init_per_testcase,term()}}`

Reason why the function was skipped.

`Fail = {error,term()} | {'EXIT',term()} | {timetrap_timeout,integer()} | {testcase_aborted,term()} | testcase_aborted_or_killed | {failed,term()} | {failed,{Mod,end_per_testcase,term()}}`

Reason why the function failed.

Args = [tuple()]

Normally Args = [Config]

This function is called when a test case, or a configuration function, is finished. It is normally called on the process where the function `Mod:Func` has been executing, but if not, the pid of the test case process is passed with the `Status` argument.

Typical use of the `end_tc/3` function can be to clean up after `init_tc/3`.

If `Func` is a test case, it is possible to analyse the value of `Result` to verify that `init_per_testcase/2` and `end_per_testcase/2` executed successfully.

It is possible with `end_tc/3` to fail an otherwise successful test case, by returning `{fail,ReasonToFail}`. The test case `Func` will be logged as failed with the provided term as reason.

```
report(What,Data) -> ok
```

Types:

What = atom()

Data = term()

This function is called in order to keep the framework up-to-date with the progress of the test. This is useful e.g. if the framework implements a GUI where the progress information is constantly updated. The following can be reported:

```
What = tests_start, Data = {Name,NumCases}
What = loginfo, Data = [{topdir,TestRootDir},{rundir,CurrLogDir}]
What = tests_done, Data = {Ok,Failed,{UserSkipped,AutoSkipped}}
What = tc_start, Data = {{Mod,Func},TCLogFile}
What = tc_done, Data = {Mod,Func,Result}
What = tc_user_skip, Data = {Mod,Func,Comment}
What = tc_auto_skip, Data = {Mod,Func,Comment}
What = framework_error, Data = {{FWMod,FWFunc},Error}
```

```
error_notification(Mod, Func, Args, Error) -> ok
```

Types:

Mod = atom()

Test suite name.

Func = atom()

Name of test case or configuration function.

Args = [tuple()]

Normally Args = [Config]

Error = {Reason,Location}

Reason = term()

Reason for termination.

Location = unknown | [{Mod,Func,Line}]

Last known position in Mod before termination.

Line = integer()

Line number in file Mod.erl.

This function is called as the result of function `Mod:Func` failing with Reason at Location. The function is intended mainly to aid specific logging or error handling in the framework application. Note that for Location to have relevant values (i.e. other than unknown), the `line` macro or `test_server_line` parse transform must be used. For details, please see the section about test suite line numbers in the `test_server` reference manual page.

`warn(What) -> boolean()`

Types:

`What = processes | nodes`

The test server checks the number of processes and nodes before and after the test is executed. This function is a question to the framework if the test server should warn when the number of processes or nodes has changed during the test execution. If `true` is returned, a warning will be written in the test case minor log file.

`target_info() -> InfoStr`

Types:

`InfoStr = string() | ""`

The test server will ask the framework for information about the test target system and print InfoStr in the test case log file below the host information.

test_server

Erlang module

The `test_server` module aids the test suite author by providing various support functions. The supported functionality includes:

- Logging and timestamping
- Capturing output to stdout
- Retrieving and flushing the message queue of a process
- Watchdog timers, process sleep, time measurement and unit conversion
- Private scratch directory for all test suites
- Start and stop of slave- or peer nodes

For more information on how to write test cases and for examples, please see the Test Server User's Guide.

TEST SUITE SUPPORT FUNCTIONS

The following functions are supposed to be used inside a test suite.

Exports

`os_type() -> OSType`

Types:

`OSType = term()`

This is the same as returned from `os:type/0`

This function can be called on controller or target node, and it will always return the OS type of the target node.

`fail()`
`fail(Reason)`

Types:

`Reason = term()`

The reason why the test case failed.

This will make the test suite fail with a given reason, or with `suite_failed` if no reason was given. Use this function if you want to terminate a test case, as this will make it easier to read the log- and HTML files. `Reason` will appear in the comment field in the HTML log.

`timetrap(Timout) -> Handle`

Types:

`Timeout = integer() | {hours,H} | {minutes,M} | {seconds,S}`
`H = M = S = integer()`
`Pid = pid()`

The process that is to be timetrapped (`self()`by default)

Sets up a time trap for the current process. An expired timetrap kills the process with reason `timetrap_timeout`. The returned handle is to be given as argument to `timetrap_cancel` before the timetrap expires. If `Timeout` is an integer, it is expected to be milliseconds.

> **Note:**
>
> If the current process is trapping exits, it will not be killed by the exit signal with reason `timetrap_timeout`. If this happens, the process will be sent an exit signal with reason `kill` 10 seconds later which will kill the process. Information about the timetrap timeout will in this case not be found in the test logs. However, the error_logger will be sent a warning.

```
timetrap_cancel(Handle) -> ok
```

Types:

`Handle = term()`
Handle returned from `timetrap`

This function cancels a timetrap. This must be done before the timetrap expires.

```
timetrap_scale_factor() -> ScaleFactor
```

Types:

`ScaleFactor = integer()`

This function returns the scale factor by which all timetraps are scaled. It is normally 1, but can be greater than 1 if the test_server is running `cover`, using a larger amount of scheduler threads than the amount of logical processors on the system, running under purify, valgrind or in a debug-compiled emulator. The scale factor can be used if you need to scale you own timeouts in test cases with same factor as the test_server uses.

```
sleep(MSecs) -> ok
```

Types:

`MSecs = integer() | float() | infinity`
The number of milliseconds to sleep

This function suspends the calling process for at least the supplied number of milliseconds. There are two major reasons why you should use this function instead of `timer:sleep`, the first being that the module `timer` may be unavailable at the time the test suite is run, and the second that it also accepts floating point numbers.

```
adjusted_sleep(MSecs) -> ok
```

Types:

`MSecs = integer() | float() | infinity`
The default number of milliseconds to sleep

This function suspends the calling process for at least the supplied number of milliseconds. The function behaves the same way as `test_server:sleep/1`, only `MSecs` will be multiplied by the 'multiply_timetraps' value, if set, and also automatically scaled up if 'scale_timetraps' is set to true (which it is by default).

```
hours(N) -> MSecs
minutes(N) -> MSecs
seconds(N) -> MSecs
```

Types:

`N = integer()`
Value to convert to milliseconds.

Theese functions convert `N` number of hours, minutes or seconds into milliseconds.

Use this function when you want to `test_server:sleep/1` for a number of seconds, minutes or hours(!).

```
format(Format) -> ok
format(Format, Args)
format(Pri, Format)
format(Pri, Format, Args)
```

Types:

`Format = string()`

Format as described for `io_:format`.

`Args = list()`

List of arguments to format.

Formats output just like `io:format` but sends the formatted string to a logfile. If the urgency value, `Pri`, is lower than some threshold value, it will also be written to the test person's console. Default urgency is 50, default threshold for display on the console is 1.

Typically, the test person don't want to see everything a test suite outputs, but is merely interested in if the test cases succeeded or not, which the test server tells him. If he would like to see more, he could manually change the threshold values by using the `test_server_ctrl:set_levels/3` function.

```
capture_start() -> ok
capture_stop() -> ok
capture_get() -> list()
```

These functions makes it possible to capture all output to stdout from a process started by the test suite. The list of characters captured can be purged by using `capture_get`.

```
messages_get() -> list()
```

This function will empty and return all the messages currently in the calling process' message queue.

```
timecall(M, F, A) -> {Time, Value}
```

Types:

`M = atom()`

The name of the module where the function resides.

`F = atom()`

The name of the function to call in the module.

`A = list()`

The arguments to supply the called function.

`Time = integer()`

The number of seconds it took to call the function.

`Value = term()`

Value returned from the called function.

This function measures the time (in seconds) it takes to call a certain function. The function call is *not* caught within a catch.

```
do_times(N, M, F, A) -> ok
do_times(N, Fun)
```

Types:

N = integer()

Number of times to call MFA.

M = atom()

Module name where the function resides.

F = atom()

Function name to call.

A = list()

Arguments to M:F.

Calls MFA or Fun N times. Useful for extensive testing of a sensitive function.

```
m_out_of_n(M, N, Fun) -> ok | exit({m_out_of_n_failed, {R,left_to_do}}
```

Types:

N = integer()

Number of times to call the Fun.

M = integer()

Number of times to require a successful return.

Repeatedly evaluates the given function until it succeeds (doesn't crash) M times. If, after N times, M successful attempts have not been accomplished, the process crashes with reason {m_out_of_n_failed, {R,left_to_do}}, where R indicates how many cases that was still to be successfully completed.

For example:

`m_out_of_n(1,4,fun() -> tricky_test_case() end)`
Tries to run tricky_test_case() up to 4 times, and is happy if it succeeds once.

`m_out_of_n(7,8,fun() -> clock_sanity_check() end)`
Tries running clock_sanity_check() up to 8 times,and allows the function to fail once. This might be useful if clock_sanity_check/0 is known to fail if the clock crosses an hour boundary during the test (and the up to 8 test runs could never cross 2 boundaries)

```
call_crash(M, F, A) -> Result
call_crash(Time, M, F, A) -> Result
call_crash(Time, Crash, M, F, A) -> Result
```

Types:

Result = ok | exit(call_crash_timeout) | exit({wrong_crash_reason, Reason})

Crash = term()

Crash return from the function.

Time = integer()

Timeout in milliseconds.

M = atom()

Module name where the function resides.

F = atom()

Function name to call.

A = list()

Arguments to M:F.

Spawns a new process that calls MFA. The call is considered successful if the call crashes with the gives reason (`Crash`) or any reason if not specified. The call must terminate within the given time (default `infinity`), or it is considered a failure.

`temp_name(Stem) -> Name`

Types:

Stem = string()

Returns a unique filename starting with `Stem` with enough extra characters appended to make up a unique filename. The filename returned is guaranteed not to exist in the filesystem at the time of the call.

`break(Comment) -> ok`

Types:

Comment = string()

`Comment` is a string which will be written in the shell, e.g. explaining what to do.

This function will cancel all timetraps and pause the execution of the test case until the user executes the `continue/0` function. It gives the user the opportunity to interact with the erlang node running the tests, e.g. for debugging purposes or for manually executing a part of the test case.

When the `break/1` function is called, the shell will look something like this:

```
--- SEMIAUTOMATIC TESTING ---
The test case executes on process <0.51.0>


"Here is a comment, it could e.g. instruct to pull out a card"


----------------------------

Continue with --> test_server:continue().
```

The user can now interact with the erlang node, and when ready call `test_server:continue().`

Note that this function can not be used if the test is executed with `ts:run/0/1/2/3/4` in `batch` mode.

`continue() -> ok`

This function must be called in order to continue after a test case has called `break/1`.

`run_on_shielded_node(Fun, CArgs) -> term()`

Types:

Fun = function() (arity 0)

Function to execute on the shielded node.

CArg = string()

Extra command line arguments to use when starting the shielded node.

`Fun` is executed in a process on a temporarily created hidden node with a proxy for communication with the test server node. The node is called a shielded node (should have been called a shield node). If `Fun` is successfully executed,

the result is returned. A peer node (see `start_node/3`) started from the shielded node will be shielded from test server node, i.e. they will not be aware of each other. This is useful when you want to start nodes from earlier OTP releases than the OTP release of the test server node.

Nodes from an earlier OTP release can normally not be started if the test server hasn't been started in compatibility mode (see the `+R` flag in the `erl(1)` documentation) of an earlier release. If a shielded node is started in compatibility mode of an earlier OTP release than the OTP release of the test server node, the shielded node can start nodes of an earlier OTP release.

> **Note:**
>
> You *must* make sure that nodes started by the shielded node never communicate directly with the test server node.

> **Note:**
>
> Slave nodes always communicate with the test server node; therefore, *never* start *slave nodes* from the shielded node, *always* start *peer nodes*.

```
start_node(Name, Type, Options) -> {ok, Node} | {error, Reason}
```

Types:

`Name = atom() | string()`

Name of the slavenode to start (as given to -sname or -name)

`Type = slave | peer`

The type of node to start.

`Options = [{atom(), term()]`

Tuplelist of options

This functions starts a node, possibly on a remote machine, and guarantees cross architecture transparency. Type is set to either `slave` or `peer`.

`slave` means that the new node will have a master, i.e. the slave node will terminate if the master terminates, TTY output produced on the slave will be sent back to the master node and file I/O is done via the master. The master is normally the target node unless the target is itself a slave.

`peer` means that the new node is an independent node with no master.

`Options` is a tuplelist which can contain one or more of

`{remote, true}`
: Start the node on a remote host. If not specified, the node will be started on the local host (with some exceptions, as for the case of VxWorks, where all nodes are started on a remote host). Test cases that require a remote host will fail with a reasonable comment if no remote hosts are available at the time they are run.

`{args, Arguments}`
: Arguments passed directly to the node. This is typically a string appended to the command line.

`{wait, false}`
: Don't wait until the node is up. By default, this function does not return until the node is up and running, but this option makes it return as soon as the node start command is given..
Only valid for peer nodes

`{fail_on_error, false}`
: Returns `{error, Reason}` rather than failing the test case.

Only valid for peer nodes. Note that slave nodes always act as if they had `fail_on_error=false`

`{erl, ReleaseList}`

Use an Erlang emulator determined by ReleaseList when starting nodes, instead of the same emulator as the test server is running. ReleaseList is a list of specifiers, where a specifier is either {release, Rel}, {prog, Prog}, or 'this'. Rel is either the name of a release, e.g., "r12b_patched" or 'latest'. 'this' means using the same emulator as the test server. Prog is the name of an emulator executable. If the list has more than one element, one of them is picked randomly. (Only works on Solaris and Linux, and the test server gives warnings when it notices that nodes are not of the same version as itself.)

When specifying this option to run a previous release, use `is_release_available/1` function to test if the given release is available and skip the test case if not.

In order to avoid compatibility problems (may not appear right away), use a shielded node (see `run_on_shielded_node/2`) when starting nodes from different OTP releases than the test server.

`{cleanup, false}`

Tells the test server not to kill this node if it is still alive after the test case is completed. This is useful if the same node is to be used by a group of test cases.

`{env, Env}`

`Env` should be a list of tuples `{Name, Val}`, where `Name` is the name of an environment variable, and `Val` is the value it is to have in the started node. Both `Name` and `Val` must be strings. The one exception is `Val` being the atom `false` (in analogy with `os:getenv/1`), which removes the environment variable. Only valid for peer nodes. Not available on VxWorks.

`{start_cover, false}`

By default the test server will start cover on all nodes when the test is run with code coverage analysis. To make sure cover is not started on a new node, set this option to `false`. This can be necessary if the connection to the node at some point will be broken but the node is expected to stay alive. The reason is that a remote cover node can not continue to run without its main node. Another solution would be to explicitly stop cover on the node before breaking the connection, but in some situations (if old code resides in one or more processes) this is not possible.

`stop_node(NodeName) -> bool()`

Types:

`NodeName = term()`

Name of the node to stop

This functions stops a node previously started with `start_node/3`. Use this function to stop any node you start, or the test server will produce a warning message in the test logs, and kill the nodes automatically unless it was started with the `{cleanup, false}` option.

`is_commercial() -> bool()`

This function test whether the emulator is commercially supported emulator. The tests for a commercially supported emulator could be more stringent (for instance, a commercial release should always contain documentation for all applications).

`is_release_available(Release) -> bool()`

Types:

`Release = string() | atom()`

Release to test for

This function test whether the release given by `Release` (for instance, "r12b_patched") is available on the computer that the test_server controller is running on. Typically, you should skip the test case if not.

Caution: This function may not be called from the `suite` clause of a test case, as the test_server will deadlock.

```
is_native(Mod) -> bool()
```

Types:

`Mod = atom()`

A module name

Checks whether the module is natively compiled or not

```
app_test(App) -> ok | test_server:fail()
app_test(App,Mode)
```

Types:

`App = term()`

The name of the application to test

`Mode = pedantic | tolerant`

Default is pedantic

Checks an applications .app file for obvious errors. The following is checked:

- required fields
- that all modules specified actually exists
- that all requires applications exists
- that no module included in the application has export_all
- that all modules in the ebin/ dir is included (If `Mode==tolerant` this only produces a warning, as all modules does not have to be included)

```
comment(Comment) -> ok
```

Types:

`Comment = string()`

The given String will occur in the comment field of the table on the HTML result page. If called several times, only the last comment is printed. comment/1 is also overwritten by the return value {comment,Comment} from a test case or by fail/1 (which prints Reason as a comment).

TEST SUITE EXPORTS

The following functions must be exported from a test suite module.

Exports

```
all(suite) -> TestSpec | {skip, Comment}
```

Types:

`TestSpec = list()`

`Comment = string()`

This comment will be printed on the HTML result page

This function must return the test specification for the test suite module. The syntax of a test specification is described in the Test Server User's Guide.

```
init_per_suite(Config0) -> Config1 | {skip, Comment}
```

Types:

`Config0 = Config1 = [tuple()]`

Comment = string()

Describes why the suite is skipped

This function is called before all other test cases in the suite. `Config` is the configuration which can be modified here. Whatever is returned from this function is given as `Config` to the test cases.

If this function fails, all test cases in the suite will be skipped.

```
end_per_suite(Config) -> void()
```

Types:

Config = [tuple()]

This function is called after the last test case in the suite, and can be used to clean up whatever the test cases have done. The return value is ignored.

```
init_per_testcase(Case, Config0) -> Config1 | {skip, Comment}
```

Types:

Case = atom()

Config0 = Config1 = [tuple()]

Comment = string()

Describes why the test case is skipped

This function is called before each test case. The `Case` 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.

```
end_per_testcase(Case, Config) -> void()
```

Types:

Case = atom()

Config = [tuple()]

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.

```
Case(doc) -> [Decription]
Case(suite) -> [] | TestSpec | {skip, Comment}
Case(Config) -> {skip, Comment} | {comment, Comment} | Ok
```

Types:

Description = string()

Short description of the test case

TestSpec = list()

Comment = string()

This comment will be printed on the HTML result page

Ok = term()

Config = [tuple()]

Elements from the Config parameter can be read with the ?config macro, see section about test suite support macros

The *documentation clause* (argument `doc`) can be used for automatic generation of test documentation or test descriptions.

The *specification clause* (argument `spec`) shall return an empty list, the test specification for the test case or `{skip,Comment}`. The syntax of a test specification is described in the Test Server User's Guide.

Note that the specification clause always is executed on the controller host.

The *execution clause* (argument `Config`) is only called if the specification clause returns an empty list. The execution clause is the real test case. Here you must call the functions you want to test, and do whatever you need to check the result. If something fails, make sure the process crashes or call `test_server:fail/0/1` (which also will cause the process to crash).

You can return `{skip,Comment}` if you decide not to run the test case after all, e.g. if it is not applicable on this platform.

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

If the execution clause returns anything else, it is considered a success, unless it is `{'EXIT',Reason}` or `{'EXIT',Pid,Reason}` which can't be distinguished from a crash, and thus will be considered a failure.

A *conf test case* is a group of test cases with an init and a cleanup function. The init and cleanup functions are also test cases, but they have special rules:

- They do not need a specification clause.
- They must always have the execution clause.
- They must return the `Config` parameter, a modified version of it or `{skip,Comment}` from the execution clause.
- The cleanup function may also return a tuple `{return_group_result,Status}`, which is used to return the status of the conf case to Test Server and/or to a conf case on a higher level. (`Status = ok | skipped | failed`).
- `init_per_testcase` and `end_per_testcase` are not called before and after these functions.

TEST SUITE LINE NUMBERS

If a test case fails, the test server can report the exact line number at which it failed. There are two ways of doing this, either by using the `line` macro or by using the `test_server_line` parse transform.

The `line` macro is described under TEST SUITE SUPPORT MACROS below. The `line` macro will only report the last line executed when a test case failed.

The `test_server_line` parse transform is activated by including the headerfile `test_server_line.hrl` in the test suite. When doing this, it is important that the `test_server_line` module is in the code path of the erlang node compiling the test suite. The parse transform will report a history of a maximum of 10 lines when a test case fails. Consecutive lines in the same function are not shown.

The attribute `-no_lines(FuncList).` can be used in the test suite to exclude specific functions from the parse transform. This is necessary e.g. for functions that are executed on old (i.e. <R10B) OTP releases. `FuncList = [{Func,Arity}]`.

If both the `line` macro and the parse transform is used in the same module, the parse transform will overrule the macro.

TEST SUITE SUPPORT MACROS

There are some macros defined in the `test_server.hrl` that are quite useful for test suite programmers:

The *line* macro, is quite essential when writing test cases. It tells the test server exactly what line of code that is being executed, so that it can report this line back if the test case fails. Use this macro at the beginning of every test case line of code.

The *config* macro, is 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 from the test server.

Possible configuration variables include:

- `data_dir` - Data file directory.
- `priv_dir` - Scratch file directory.
- `nodes` - Nodes specified in the spec file
- `nodenames` - Generated nodenames.
- Whatever added by conf test cases or `init_per_testcase/2`

Examples of the `line` and `config` macros can be seen in the Examples chapter in the user's guide.

If the `line_trace` macro is defined, you will get a timestamp (`erlang:now()`) in your minor log for each `line` macro in your suite. This way you can at any time see which line is currently being executed, and when the line was called.

The `line_trace` macro can also be used together with the `test_server_line` parse transform described above. A timestamp will then be written for each line in the suite, except for functions stated in the `-no_lines` attribute.

The `line_trace` macro can e.g. be defined as a compile option, like this:

```
erlc -W -Dline_trace my_SUITE.erl
```