NAME

ct - Main user interface for the Common Test framework.

DESCRIPTION

Main user interface for the *Common Test* framework.

This module implements the command-line interface for running tests and basic functions for *Common Test* case issues, such as configuration and logging.

Test Suite Support Macros

The *config* macro is defined in *ct.hrl*. This macro is to be used to retrieve information from the *Config* variable sent to all test cases. It is used with two arguments; the first is the name of the configuration variable to retrieve, the second is the *Config* variable supplied to the test case.

Possible configuration variables include:

* *data_dir* - Data file directory
* *priv_dir* - Scratch file directory
* Whatever added by *init_per_suite/1* or *init_per_testcase/2* in the test suite.

DATA TYPES

handle() = pid()

The identity (handle) of a connection.

config_key() = atom()

A configuration key which exists in a configuration file

target_name() = atom()

A name and association to configuration data introduced through a require statement, or a call to *ct:require/2*, for example, *ct:require(mynodename,{node,[telnet]})*.

key_or_name() = config_key() | target_name()

conn_log_options() = [conn_log_option()]

Options that can be given to the *cth_conn_log* hook, which is used for logging of NETCONF and Telnet connections. See ct_netconfc or ct_telnet for description and examples of how to use this hook.

conn_log_option() = {log_type,conn_log_type()} | {hosts,[key_or_name()]}

conn_log_type() = raw | pretty | html | silent

conn_log_mod() = ct_netconfc | ct_telnet

EXPORTS

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

Types:

Reason = term()
ErrorReason = no_testcase_running | parallel_group

Aborts the currently executing test case. The user must know with certainty which test case is currently executing. The function is therefore only safe to call from a function that has been called (or synchronously invoked) by the test case.

Reason, the reason for aborting the test case, is printed in the test case log.

add_config(Callback, Config) -> ok | {error, Reason}

Types:

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

Loads configuration variables using the specified callback module and configuration string. The callback module is to be either loaded or present in the code path. Loaded configuration variables can later be removed using function *ct:remove_config/2*.

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

Types:

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

Cancels any active timetrap and pauses the execution of the current test case until the user calls function *continue/0*. The user can then interact with the Erlang node running the tests, for example, for debugging purposes or for manually executing a part of the test case. If a parallel group is executing, *ct:break/2* is to be called instead.

A cancelled timetrap is not automatically reactivated after the break, but must be started explicitly with *ct:timetrap/1*.

In order for the break/continue functionality to work, *Common Test* must release the shell process controlling *stdin*. This is done by setting start option *release_shell* to *true*. For details, see section Running Tests from the Erlang Shell or from an Erlang Program in the User’s Guide.

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

Types:

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

Works the same way as *ct:break/1*, only argument *TestCase* makes it possible to pause a test case executing in a parallel group. Function *ct:continue/1* is to be used to resume execution of *TestCase*.

For details, see *ct:break/1*.

capture_get() -> ListOfStrings

Types:

ListOfStrings = [string()]

Equivalent to ct:capture_get([default]).

capture_get(ExclCategories) -> ListOfStrings

Types:

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

Returns and purges the list of text strings buffered during the latest session of capturing printouts to *stdout*. Log categories that are to be ignored in *ListOfStrings* can be specified with *ExclCategories*. If *ExclCategories = []*, no filtering takes place.

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

capture_start() -> ok

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

See also *ct:capture_get/1*, *ct:capture_stop/0*.

capture_stop() -> ok

Stops capturing text strings (a session started with *capture_start/0*).

See also *ct:capture_get/1*, *ct:capture_start/0*.

comment(Comment) -> ok

Types:

Comment = term()

Prints the specified *Comment* in the comment field in the table on the test suite result page.

If called several times, only the last comment is printed. The test case return value *{comment,Comment}* overwrites the string set by this function.

comment(Format, Args) -> ok

Types:

Format = string()
Args = list()

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

Arguments *Format* and *Args* are used in a call to *io_lib:format/2* to create the comment string. The behavior of *comment/2* is otherwise the same as function *ct:comment/1*.

continue() -> ok

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

continue(TestCase) -> ok

Types:

TestCase = atom()

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

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

Types:

EncryptFileName = string()
TargetFileName = string()
Reason = term()

Decrypts *EncryptFileName*, previously generated with *ct:encrypt_config_file/2,3*. The original file contents is saved in the target file. The encryption key, a string, must be available in a text file named *.ct_config.crypt*, either in the current directory, or the home directory of the user (it is searched for in that order).

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

Types:

EncryptFileName = string()
TargetFileName = string()
KeyOrFile = {key, string()} | {file, string()}

Reason = term()

Decrypts *EncryptFileName*, previously generated with *ct:encrypt_config_file/2,3*. The original file contents is saved in the target file. The key must have the same value as that used for encryption.

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

Types:

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

Encrypts the source configuration file with DES3 and saves the result in file *EncryptFileName*. The key, a string, must be available in a text file named *.ct_config.crypt*, either in the current directory, or the home directory of the user (it is searched for in that order).

For information about using encrypted configuration files when running tests, see section Encrypted Configuration Files in the User's Guide.

For details on DES3 encryption/decryption, see application *Crypto*.

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

Types:

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

Encrypts the source configuration file with DES3 and saves the result in the target file *EncryptFileName*. The encryption key to use is either the value in *{key,Key}* or the value stored in the file specified by *{file,File}*.

For information about using encrypted configuration files when running tests, see section Encrypted Configuration Files in the User's Guide.

For details on DES3 encryption/decryption, see application *Crypto*.

fail(Reason) -> ok

Types:

Reason = term()

Terminates a test case with the specified error *Reason*.

fail(Format, Args) -> ok

Types:

Format = string()
Args = list()

Terminates a test case with an error message specified by a format string and a list of values (used as arguments to *io_lib:format/2*).

get_config(Required) -> Value

Equivalent to *ct:get_config(Required, undefined, [])*.

get_config(Required, Default) -> Value

Equivalent to *ct:get_config(Required, Default, [])*.

get_config(Required, Default, Opts) -> ValueOrElement

Types:

Required = KeyOrName | {KeyOrName, SubKey} | {KeyOrName, SubKey, SubKey}
KeyOrName = atom()
SubKey = atom()
Default = term()
Opts = [Opt] | []
Opt = element | all
ValueOrElement = term() | Default

Reads configuration data values.

Returns the matching values or configuration elements, given a configuration variable key or its associated name (if one has been specified with *ct:require/2* or a *require* statement).

Example:

Given the following configuration file:

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

Then:

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

If a configuration variable key has been associated with a name (by *ct:require/2* or a *require* statement), the name can be used instead of the key to read the value:

```
 ct:require(myuser,{unix,user}) -> ok.
 ct:get_config(myuser,Default) -> [{username,Username}, {password,Password}]
```

If a configuration variable is defined in multiple files, use option *all* to access all possible values. The values are returned in a list. The order of the elements corresponds to the order that the configuration files were specified at startup.

If configuration elements (key-value tuples) are to be returned as result instead of values, use option *element*. The returned elements are then on the form *{Required,Value}*.

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

get_event_mgr_ref() -> EvMgrRef

Types:

EvMgrRef = atom()

Gets a reference to the *Common Test* event manager. The reference can be used to, for example, add a user-specific event handler while tests are running.

Example:

```
 gen_event:add_handler(ct:get_event_mgr_ref(), my_ev_h, [])
```

get_progname() -> string()

Returns the command used to start this Erlang instance. If this information could not be found, the string *"no_prog_name"* is returned.

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

Types:

TestStatus = [StatusElem]
StatusElem = {current, TestCaseInfo} | {successful, Successful} | {failed, Failed} | {skipped, Skipped} | {total, Total}
TestCaseInfo = {Suite, TestCase} | [{Suite, TestCase}]
Suite = atom()
TestCase = atom()
Successful = integer()
Failed = integer()
Skipped = {UserSkipped, AutoSkipped}
UserSkipped = integer()
AutoSkipped = integer()
Total = integer()
Reason = term()

Returns status of ongoing test. The returned list contains information about which test case is executing (a list of cases when a parallel test case group is executing), as well as counters for successful, failed, skipped, and total test cases so far.

get_target_name(Handle) -> {ok, TargetName} | {error, Reason}

Types:

Handle = handle()
TargetName = target_name()

Returns the name of the target that the specified connection belongs to.

get_testspec_terms() -> TestSpecTerms | undefined

Types:

TestSpecTerms = [{Tag, Value}]
Value = [term()]

Gets a list of all test specification terms used to configure and run this test.

get_testspec_terms(Tags) -> TestSpecTerms | undefined

Types:

Tags = [Tag] | Tag
Tag = atom()
TestSpecTerms = [{Tag, Value}] | {Tag, Value}
Value = [{Node, term()}] | [term()]
Node = atom()

Reads one or more terms from the test specification used to configure and run this test. *Tag* is any valid test specification tag, for example, *label*, *config*, or *logdir*. User-specific terms are also available to read if option *allow_user_terms* is set.

All value tuples returned, except user terms, have the node name as first element.

To read test terms, use *Tag = tests* (rather than *suites*, *groups*, or *cases*). *Value* is then the list of *all* tests on the form *[{Node,Dir,[{TestSpec,GroupsAndCases1},...]},...]*, where *GroupsAndCases = [{Group,[Case]}] | [Case]*.

get_timetrap_info() -> {Time, {Scaling,ScaleVal}}

Types:

Time = integer() | infinity
Scaling = true | false
ScaleVal = integer()

Reads information about the timetrap set for the current test case. *Scaling* indicates if *Common Test* will attempt to compensate timetraps automatically for runtime delays introduced by, for example, tools like cover. *ScaleVal* is the value of the current scaling multiplier (always 1 if scaling is disabled). Note the *Time* is not the scaled result.

get_verbosity(Category) -> Level | undefined

Types:

Category = default | atom()
Level = integer()

This function returns the verbosity level for the specified logging category. See the User's Guide for details. Use the value *default* to read the general verbosity level.

install(Opts) -> ok | {error, Reason}

Types:

Opts = [Opt]
Opt = {config, ConfigFiles} | {event_handler, Modules} | {decrypt, KeyOrFile}
ConfigFiles = [ConfigFile]
ConfigFile = string()
Modules = [atom()]
KeyOrFile = {key, Key} | {file, KeyFile}
Key = string()
KeyFile = string()

Installs configuration files and event handlers.

Run this function once before the first test.

Example:

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

This function is automatically run by program *ct_run*.

listenv(Telnet) -> [Env]

Types:

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

Performs command *listenv* on the specified Telnet connection and returns the result as a list of key-value pairs.

log(Format) -> ok

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

log(X1, X2) -> ok

Types:

X1 = Category | Importance | Format
X2 = Format | FormatArgs

Equivalent to *ct:log(Category, Importance, Format, FormatArgs, [])*.

log(X1, X2, X3) -> ok

Types:

X1 = Category | Importance
X2 = Importance | Format
X3 = Format | FormatArgs | Opts

Equivalent to *ct:log(Category, Importance, Format, FormatArgs, Opts)*.

log(X1, X2, X3, X4) -> ok

Types:

X1 = Category | Importance
X2 = Importance | Format
X3 = Format | FormatArgs
X4 = FormatArgs | Opts

Equivalent to *ct:log(Category, Importance, Format, FormatArgs, Opts)*.

log(Category, Importance, Format, FormatArgs, Opts) -> ok

Types:

Category = atom()
Importance = integer()
Format = string()
FormatArgs = list()
Opts = [Opt]
Opt = {heading,string()} | no_css | esc_chars

Prints from a test case to the log file.

This function is meant for printing a string directly from a test case to the test case log file.

Default *Category* is *default*, default *Importance* is *?STD_IMPORTANCE*, and default value for *FormatArgs* is *[]*.

For details on *Category*, *Importance* and the *no_css* option, see section Logging - Categories and Verbosity Levels in the User's Guide.

Common Test will not escape special HTML characters (<, > and &) in the text printed with this function, unless the *esc_chars* option is used.

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

Types:

Reason = term()

If the test is started with option *create_priv_dir* set to *manual_per_tc*, in order for the test case to use the private directory, it must first create it by calling this function.

notify(Name, Data) -> ok

Types:

Name = atom()
Data = term()

Sends an asynchronous notification of type *Name* with *Data*to the Common Test event manager. This can later be caught by any installed event manager.

See also *gen_event(3)*.

pal(Format) -> ok

Equivalent to *ct:pal(default, 50, Format, [], [])*.

pal(X1, X2) -> ok

Types:

X1 = Category | Importance | Format
X2 = Format | FormatArgs

Equivalent to *ct:pal(Category, Importance, Format, FormatArgs, [])*.

pal(X1, X2, X3) -> ok

Types:

X1 = Category | Importance
X2 = Importance | Format
X3 = Format | FormatArgs | Opts

Equivalent to *ct:pal(Category, Importance, Format, FormatArgs, Opts)*.

pal(X1, X2, X3, X4) -> ok

Types:

X1 = Category | Importance

X2 = Importance | Format
X3 = Format | FormatArgs
X4 = FormatArgs | Opts

Equivalent to *ct:pal(Category, Importance, Format, FormatArgs, Opts)*.

pal(Category, Importance, Format, FormatArgs, Opts) -> ok

Types:

Category = atom()
Importance = integer()
Format = string()
FormatArgs = list()
Opts = [Opt]
Opt = {heading,string()} | no_css

Prints and logs from a test case.

This function is meant for printing a string from a test case, both to the test case log file and to the console.

Default *Category* is *default*, default *Importance* is *?STD_IMPORTANCE*, and default value for *FormatArgs* is *[]*.

For details on *Category* and *Importance*, see section Logging - Categories and Verbosity Levels in the User's Guide.

Note that special characters in the text (<, > and &) will be escaped by Common Test before the text is printed to the log file.

parse_table(Data) -> {Heading, Table}

Types:

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

Parses the printout from an SQL table and returns a list of tuples.

The printout to parse is typically the result of a *select* command in SQL. The returned *Table* is a list of tuples, where each tuple is a row in the table.

Heading is a tuple of strings representing the headings of each column in the table.

print(Format) -> ok

Equivalent to *ct:print(default, 50, Format, [], [])*.

print(X1, X2) -> ok

Types:

X1 = Category | Importance | Format
X2 = Format | FormatArgs

Equivalent to *ct:print(Category, Importance, Format, FormatArgs, [])*.

print(X1, X2, X3) -> ok

Types:

X1 = Category | Importance
X2 = Importance | Format
X3 = Format | FormatArgs | Opts

Equivalent to *ct:print(Category, Importance, Format, FormatArgs, Opts)*.

print(X1, X2, X3, X4) -> ok

Types:

X1 = Category | Importance
X2 = Importance | Format
X3 = Format | FormatArgs
X4 = FormatArgs | Opts

Equivalent to *ct:print(Category, Importance, Format, FormatArgs, Opts)*.

print(Category, Importance, Format, FormatArgs, Opts) -> ok

Types:

Category = atom()
Importance = integer()
Format = string()
FormatArgs = list()
Opts = [Opt]
Opt = {heading,string()}

Prints from a test case to the console.

This function is meant for printing a string from a test case to the console.

Default *Category* is *default*, default *Importance* is *?STD_IMPORTANCE*, and default value for *FormatArgs* is *[]*.

For details on *Category* and *Importance*, see section Logging - Categories and Verbosity Levels in the User's Guide.

reload_config(Required) -> ValueOrElement | {error, Reason}

Types:

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

Reloads configuration file containing specified configuration key.

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

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

remaining_test_procs() -> {TestProcs,SharedGL,OtherGLs}

Types:

TestProcs = [{pid(),GL}]
GL = pid()
SharedGL = pid()
OtherGLs = [pid()]

This function will return the identity of test- and group leader processes that are still running at the time of this call. *TestProcs* are processes in the system that have a Common Test IO process as group leader. *SharedGL* is the central Common Test IO process, responsible for printing to log files for configuration functions and sequentially executing test cases. *OtherGLs* are Common Test IO processes that print to log files for test cases in parallel test case groups.

The process information returned by this function may be used to locate and terminate remaining processes after tests have finished executing. The function would typically by called from Common Test Hook functions.

Note that processes that execute configuration functions or test cases are never included in *TestProcs*. It is therefore safe to use post configuration hook functions (such as post_end_per_suite, post_end_per_group, post_end_per_testcase) to terminate all processes in *TestProcs* that have the current group leader process as its group leader.

Note also that the shared group leader (*SharedGL*) must never be terminated by the user, only by Common Test. Group leader processes for parallel test case groups (*OtherGLs*) may however be terminated in post_end_per_group hook functions.

remove_config(Callback, Config) -> ok

Types:

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

Removes configuration variables (together with their aliases) that were loaded with specified callback module and configuration string.

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

Types:

Required = Key | {Key, SubKeys} | {Key, SubKey, SubKeys}

Key = atom()
SubKeys = SubKey | [SubKey]
SubKey = atom()

Checks if the required configuration is available. Arbitrarily deep tuples can be specified as *Required*. Only the last element of the tuple can be a list of *SubKey*s.

Example 1. Require the variable *myvar*:

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

In this case the configuration file must at least contain:

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

Example 2. Require key *myvar* with subkeys *sub1* and *sub2*:

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

In this case the configuration file must at least contain:

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

Example 3. Require key *myvar* with subkey *sub1* with *subsub1*:

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

In this case the configuration file must at least contain:

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

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

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

Types:

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

Checks if the required configuration is available and gives it a name. The semantics for *Required* is the same as in *ct:require/1* except that a list of *SubKey*s cannot be specified.

If the requested data is available, the subentry is associated with *Name* so that the value of the element can be read with *ct:get_config/1,2* provided *Name* is used instead of the whole *Required* term.

Example:

Require one node with a Telnet connection and an FTP connection. Name the node *a*:

```
 ok = ct:require(a,{machine,node}).
```

All references to this node can then use the node name. For example, a file over FTP is fetched like follows:

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

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

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

Note:

The behavior of this function changed radically in *Common Test* 1.6.2. To keep some backwards compatibility, it is still possible to do:
ct:require(a,{node,[telnet,ftp]}).
This associates the name *a* with the top-level *node* entry. For this to work, the configuration file must at least contain:
{node,[{telnet,IpAddr},{ftp,IpAddr}]}.

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

run(TestDirs) -> Result

Types:

TestDirs = TestDir | [TestDir]

Runs all test cases in all suites in the specified directories.

See also *ct:run/3*.

run(TestDir, Suite) -> Result

Runs all test cases in the specified suite.

See also *ct:run/3*.

run(TestDir, Suite, Cases) -> Result

Types:

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

Runs the specified test cases.

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

Suites (**_SUITE.erl*) files must be stored in *TestDir* or *TestDir/test*. All suites are compiled when the test is run.

run_test(Opts) -> Result

Types:

Opts = [OptTuples]
OptTuples = {dir, TestDirs} | {suite, Suites} | {group, Groups} | {testcase, Cases} | {spec,

TestSpecs} | {join_specs, Bool} | {label, Label} | {config, CfgFiles} | {userconfig, UserConfig} | {allow_user_terms, Bool} | {logdir, LogDir} | {silent_connections, Conns} | {stylesheet, CSSFile} | {cover, CoverSpecFile} | {cover_stop, Bool} | {step, StepOpts} | {event_handler, EventHandlers} | {include, InclDirs} | {auto_compile, Bool} | {abort_if_missing_suites, Bool} | {create_priv_dir, CreatePrivDir} | {multiply_timetraps, M} | {scale_timetraps, Bool} | {repeat, N} | {duration, DurTime} | {until, StopTime} | {force_stop, ForceStop} | {decrypt, DecryptKeyOrFile} | {refresh_logs, LogDir} | {logopts, LogOpts} | {verbosity, VLevels} | {basic_html, Bool} | {esc_chars, Bool} | {keep_logs,KeepSpec} | {ct_hooks, CTHs} | {enable_builtin_hooks, Bool} | {release_shell, Bool}
TestDirs = [string()] | string()
Suites = [string()] | [atom()] | string() | atom()
Cases = [atom()] | atom()
Groups = GroupNameOrPath | [GroupNameOrPath]
GroupNameOrPath = [atom()] | atom() | all
TestSpecs = [string()] | string()
Label = string() | atom()
CfgFiles = [string()] | string()
UserConfig = [{CallbackMod, CfgStrings}] | {CallbackMod, CfgStrings}
CallbackMod = atom()
CfgStrings = [string()] | string()
LogDir = string()
Conns = all | [atom()]
CSSFile = string()
CoverSpecFile = string()
StepOpts = [StepOpt] | []
StepOpt = config | keep_inactive
EventHandlers = EH | [EH]
EH = atom() | {atom(), InitArgs} | {[atom()], InitArgs}
InitArgs = [term()]
InclDirs = [string()] | string()
CreatePrivDir = auto_per_run | auto_per_tc | manual_per_tc
M = integer()
N = integer()
DurTime = string(HHMMSS)
StopTime = string(YYMoMoDDHHMMSS) | string(HHMMSS)
ForceStop = skip_rest | Bool
DecryptKeyOrFile = {key, DecryptKey} | {file, DecryptFile}
DecryptKey = string()
DecryptFile = string()
LogOpts = [LogOpt]

LogOpt = no_nl | no_src
VLevels = VLevel | [{Category, VLevel}]
VLevel = integer()
Category = atom()
KeepSpec = all | pos_integer()
CTHs = [CTHModule | {CTHModule, CTHInitArgs}]
CTHModule = atom()
CTHInitArgs = term()
Result = {Ok, Failed, {UserSkipped, AutoSkipped}} | TestRunnerPid | {error, Reason}
Ok = integer()
Failed = integer()
UserSkipped = integer()
AutoSkipped = integer()
TestRunnerPid = pid()
Reason = term()

Runs tests as specified by the combination of options in *Opts*. The options are the same as those used with program *ct_run*, see Run Tests from Command Line in the *ct_run* manual page.

Here a *TestDir* can be used to point out the path to a *Suite*. Option *testcase* corresponds to option *-case* in program *ct_run*. Configuration files specified in *Opts* are installed automatically at startup.

TestRunnerPid is returned if *release_shell == true*. For details, see *ct:break/1*.

Reason indicates the type of error encountered.

run_testspec(TestSpec) -> Result

Types:

TestSpec = [term()]
Result = {Ok, Failed, {UserSkipped, AutoSkipped}} | {error, Reason}
Ok = integer()
Failed = integer()
UserSkipped = integer()
AutoSkipped = integer()
Reason = term()

Runs a test specified by *TestSpec*. The same terms are used as in test specification files.

Reason indicates the type of error encountered.

set_verbosity(Category, Level) -> ok

Types:

Category = default | atom()
Level = integer()

Use this function to set, or modify, the verbosity level for a logging category. See the User's Guide for details. Use the value *default* to set the general verbosity level.

sleep(Time) -> ok

Types:

Time = {hours, Hours} | {minutes, Mins} | {seconds, Secs} | Millisecs | infinity
Hours = integer()
Mins = integer()
Secs = integer()
Millisecs = integer() | float()

This function, similar to *timer:sleep/1* in STDLIB, suspends the test case for a specified time. However, this function also multiplies *Time* with the *multiply_timetraps* value (if set) and under certain circumstances also scales up the time automatically if *scale_timetraps* is set to *true* (default is *false*).

start_interactive() -> ok

Starts *Common Test* in interactive mode.

From this mode, all test case support functions can be executed directly from the Erlang shell. The interactive mode can also be started from the OS command line with *ct_run -shell [-config File...]*.

If any functions (for example, Telnet or FTP) using "required configuration data" are to be called from the Erlang shell, configuration data must first be required with *ct:require/2*.

Example:

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

step(TestDir, Suite, Case) -> Result

Types:

Case = atom()

Steps through a test case with the debugger.

See also *ct:run/3*.

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

Types:

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

Steps through a test case with the debugger. If option *config* has been specified, breakpoints are also set on the configuration functions in *Suite*.

See also *ct:run/3*.

stop_interactive() -> ok

Exits the interactive mode.

See also *ct:start_interactive/0*.

sync_notify(Name, Data) -> ok

Types:

Name = atom()
Data = term()

Sends a synchronous notification of type *Name* with *Data* to the *Common Test* event manager. This can later be caught by any installed event manager.

See also *gen_event(3)*.

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

Types:

TestDir = string()
Suite = atom()
Testcases = list()
Reason = term()

Returns all test cases in the specified suite.

timetrap(Time) -> ok

Types:

Time = {hours, Hours} | {minutes, Mins} | {seconds, Secs} | Millisecs | infinity | Func
Hours = integer()
Mins = integer()
Secs = integer()
Millisecs = integer()
Func = {M, F, A} | function()
M = atom()
F = atom()
A = list()

Sets a new timetrap for the running test case.

If the argument is *Func*, the timetrap is triggered when this function returns. *Func* can also return a new *Time* value, which in that case is the value for the new timetrap.

userdata(TestDir, Suite) -> SuiteUserData | {error, Reason}

Types:

TestDir = string()
Suite = atom()
SuiteUserData = [term()]
Reason = term()

Returns any data specified with tag *userdata* in the list of tuples returned from *suite/0*.

userdata(TestDir, Suite, Case::GroupOrCase) -> TCUserData | {error, Reason}

Types:

TestDir = string()
Suite = atom()
GroupOrCase = {group, GroupName} | atom()
GroupName = atom()
TCUserData = [term()]
Reason = term()

Returns any data specified with tag *userdata* in the list of tuples returned from *Suite:group(GroupName)* or *Suite:Case()*.