Runtime_Tools

version 1.2

Typeset in LaTeX from SGML source using the DOCBUILDER 3.2.2 Document System.

Contents

1 Runtime_Tools Reference Manual 1
1.1 runtime_tools 4
1.2 dbg 5

Runtime_Tools Reference Manual

Short Summaries

- Application **runtime_tools** [page 4] – The Runtime tools Application
- Erlang Module **dbg** [page 5] – The Text Based Trace Facility

runtime_tools

No functions are exported.

dbg

The following functions are exported:

- `h() -> ok`
  [page 5] Give a list of available help items on standard output.
- `h(Item) -> ok`
  [page 5] Give brief help for an item.
- `p(Item) -> {ok, MatchDesc} | {error, term()}`
  [page 5] Trace messages to and from Item.
- `p(Item, Flags) -> {ok, MatchDesc} | {error, term()}`
  [page 5] Trace Item according to Flags.
- `c(Mod, Fun, Args)`
  [page 6] Evaluate `apply(M,F,Args)` with `all` trace flags set.
- `c(Mod, Fun, Args, Flags)`
  [page 6] Evaluate `apply(M,F,Args)` with `Flags` trace flags set.
- `i() -> ok`
  [page 7] Display information about all traced processes.
- `tp(Module,MatchSpec)`
  [page 7] Set pattern for traced global function calls
- `tp(Module,Function,MatchSpec)`
  [page 7] Set pattern for traced global function calls
- `tp(Module, Function, Arity, MatchSpec)`
  [page 7] Set pattern for traced global function calls

- `tp({Module, Function, Arity}, MatchSpec) -> {ok, MatchDesc} | {error, term()}`
  [page 7] Set pattern for traced global function calls
- `tpl(Module,MatchSpec)`
  [page 8] Set pattern for traced local (as well as global) function calls
- `tpl(Module,Function,MatchSpec)`
  [page 8] Set pattern for traced local (as well as global) function calls
- `tpl(Module, Function, Arity, MatchSpec)`
  [page 8] Set pattern for traced local (as well as global) function calls
- `tpl({Module, Function, Arity}, MatchSpec) -> {ok, MatchDesc} | {error, term()}`
  [page 8] Set pattern for traced local (as well as global) function calls
- `ctp(Module)`
  [page 8] Clear call trace pattern for the specified functions
- `ctp(Module, Function)`
  [page 8] Clear call trace pattern for the specified functions
- `ctp(Module, Function, Arity)`
  [page 8] Clear call trace pattern for the specified functions
- `ctp({Module, Function, Arity}) -> {ok, MatchDesc} | {error, term()}`
  [page 8] Clear call trace pattern for the specified functions
- `ctpl(Module)`
  [page 8] Clear call trace pattern for the specified functions
- `ctpl(Module, Function)`
  [page 9] Clear call trace pattern for the specified functions
- `ctpl(Module, Function, Arity)`
  [page 9] Clear call trace pattern for the specified functions
- `ctpl({Module, Function, Arity}) -> {ok, MatchDesc} | {error, term()}`
  [page 9] Clear call trace pattern for the specified functions
- `ctpg(Module)`
  [page 9] Clear call trace pattern for the specified functions
- `ctpg(Module, Function)`
  [page 9] >Clear call trace pattern for the specified functions
- `ctpg(Module, Function, Arity)`
  [page 9] >Clear call trace pattern for the specified functions
- `ctpg({Module, Function, Arity}) -> {ok, MatchDesc} | {error, term()}`
  [page 9] Clear call trace pattern for the specified functions
- `ltp() -> ok`
  [page 9] List saved match_spec's on the console.
- `dtp() -> ok`
  [page 9] Delete all saved match_spec's.
- `dtp(N) -> ok`
  [page 9] Delete a specific saved match_spec.
- `wtp(Name) -> ok | {error, IOError}`
  [page 9] Write all saved match_spec's to a file
- `rtp(Name) -> ok | {error, Error}`
  [page 10] Read saved match specifications from file.

- `n(Nodename) -> {ok, Nodename} | {error, Reason}`
  [page 10] Add a node to the list of traced nodes
- `cn(Nodename) -> ok`
  [page 10] Clear a node from the list of traced nodes.
- `ln() -> ok`
  [page 11] Show the list of traced nodes on the console.
- `tracer() -> {ok, pid()} | {error, already_started}`
  [page 11] Start a tracer server that handles trace messages.
- `tracer(Type, Data) -> {ok, pid()} | {error, Error}`
  [page 11] Start a tracer server with additional parameters
- `trace_port(Type, Parameters) -> fun()`
  [page 11] Create and returns a trace port generating *fun*
- `flush_trace_port() -> ok | {error, Reason}`
  [page 12] Flush internal data buffers in a trace driver.
- `trace_port_control(Operation) -> ok | {ok, Result}, {error, Reason}`
  [page 13] Perform a control operation on the active trace port driver.
- `trace_client(Type, Parameters) -> pid()`
  [page 13] Start a trace client that reads messages created by a trace port driver
- `trace_client(Type, Parameters, HandlerSpec) -> pid()`
  [page 14] Start a trace client that reads messages created by a trace port driver, with a user defined handler
- `stop_trace_client(Pid) -> ok`
  [page 14] Stop a trace client gracefully.
- `get_tracer() -> {ok, Tracer}`
  [page 14] Return the process or port to which all trace messages are sent.
- `stop() -> stopped`
  [page 15] Stop the `dbg` server and the tracing of all processes.

runtime_tools

Application

This chapter describes the Runtime_Tools application in OTP, which provides low footprint tracing/debugging tools suitable for inclusion in a production system.

Configuration

There are currently no configuration parameters available for this application.

SEE ALSO

application(3)

dbg

Erlang Module

This module implements a text based interface to the `trace/3` and the trace_pattern/2 BIF's. It makes it possible to trace functions, processes, and messages on text based terminals. It can be used instead of, or as complement to, the `pman` module.

The utilities are suitable to use in system testing on large systems, where other tools have too much impact on the system performance. Some primitive support for sequential tracing is also included, see the advanced topics [page 15] section.

Exports

`h() -> ok`

Gives a list of items for brief online help.

`h(Item) -> ok`

Types:

- Item = atom()

Gives a brief help text for functions in the dbg module. The available items can be listed with `dbg:h/0`

`p(Item) -> {ok, MatchDesc} | {error, term()}`

Equivalent to `p(Item, [m])`.

`p(Item, Flags) -> {ok, MatchDesc} | {error, term()}`

Types:

- MatchDesc = [MatchNum]
- MatchNum = {matched, integer()} | {matched, node(), integer()} | {matched, node(), 0, RPCError}
- RPCError = term()

Traces `Item` in accordance to the value specified by `Flags`. The variation of `Item` is listed below:

- If the `Item` is a pid(), the corresponding process is traced. If no trace port is used, the process may be a remote process (on another Erlang node). The node must be on the list of traced nodes (see [page 10] `n/1`).
- If the `Item` is the atom `all`, all processes in the system as well as all processes created hereafter are to be traced. This also affects all nodes added with the `n/1` function.

- If the `Item` is the atom `new`, no currently existing processes are affected, but every process created after the call is.This also affects all nodes added with the `n/1` function.
- If the `Item` is the atom `existing`, all existing processes are traced, but new processes will not be affected.This also affects all nodes added with the `n/1` function.
- If the `Item` is an atom other than `all`, `new` or `existing`, the process with the corresponding registered name is traced.
- If the `Item` is an integer, the process <`Item.1`> is traced.
- If the `Item` is a tuple {`X, Y, Z`}, the process <`X.Y.Z`> is traced.

`Flags` can be a single atom, or a list of flags. The available flags are:

`s (send)` Traces the messages the process sends.

`r (receive)` Traces the messages the process receives.

`m (messages)` Traces the messages the process receives and sends.

`c (call)` Traces global function calls for the process according to the trace patterns set in the system (see tp/2).

`p (proc)` Traces process related events to the process.

`sos (set on spawn)` Lets all processes created by the traced process inherit the trace flags of the traced process.

`sol (set on link)` Lets another process, `P2`, inherit the trace flags of the traced process whenever the traced process links to `P2`.

`sofs (set on first spawn)` This is the same as `sos`, but only for the first process spawned by the traced process.

`sofl (set on first link)` This is the same as `sol`, but only for the first call to `link/1` by the traced process.

`all` Sets all flags.

`clear` Clears all flags.

The list can also include any of the flags allowed in `erlang:trace/3`

The function returns either an error tuple or a tuple {`ok, List`}. The `List` consists of specifications of how many processes that matched (in the case of a pure pid() exactly 1). The specification of matched processes can be either {`matched, N`}, when only local processes matched, or {`matched, Node, N`} in the case of tracing a remote node (as well as the local). If the remote processor call,`rpc`, to a remote node fails, the `rpc` error message is delivered as a fourth argument and the number of matched processes are 0. Note that the result {ok, List} may contain a list where `rpc` calls to one or more nodes failed. The `ok` only means that some processes matched and are traced.

`c(Mod, Fun, Args)`

Equivalent to `c(Mod, Fun, Args, all)`.

`c(Mod, Fun, Args, Flags)`

Evaluates the expression `apply(Mod, Fun, Args)` with the trace flags in `Flags` set. This is a convenient way to trace processes from the Erlang shell.

`i() -> ok`

Displays information about all traced processes.

`tp(Module,MatchSpec)`

Same as tp({Module, '_', '_'}, MatchSpec)

`tp(Module,Function,MatchSpec)`

Same as tp({Module, Function, '_'}, MatchSpec)

`tp(Module, Function, Arity, MatchSpec)`

Same as tp({Module, Function, Arity}, MatchSpec)

`tp({Module, Function, Arity}, MatchSpec) -> {ok, MatchDesc} | {error, term()}`

Types:

- Module = atom() | '_'
- Function = atom() | '_'
- Arity = integer() |'_'
- MatchSpec = integer() | [] | match_spec()
- MatchDesc = [MatchInfo]
- MatchInfo = {saved, integer()} | MatchNum <V>MatchNum = {matched, integer()} | {matched, node(), integer()} | {matched, node(), 0, RPCError}

This function enables call trace for one or more functions. All exported functions matching the `{Module, Function, Arity}` argument will be concerned, but the `match_spec()` may further narrow down the set of function calls generating trace messages.

For a description of the `match_spec()` syntax, please turn to the *User's guide* part of the online documentation for the runtime system (*erts*). The chapter *Match Specification in Erlang* explains the general match specification "language".

The Module, Function and/or Arity parts of the tuple may be specified as the atom `'_'` which is a "wild-card" matching all modules/functions/arities. Note, if the Module is specified as `'_'`, the Function and Arity parts have to be specified as '_' too. The same holds for the Functions relation to the Arity.

All nodes added with `n/1` will be affected by this call, and if Module is not `'_'` the module will be loaded on all nodes.

The function returns either an error tuple or a tuple `{ok, List}`. The `List` consists of specifications of how many functions that matched, in the same way as the processes are presented in the return value of `p/2`.

There may be a tuple `{saved, N}` in the return value, if the MatchSpec is other than []. The integer `N` may then be used in subsequent calls to this function and will stand as an "alias" for the given expression (see also ltp/0 below).

If an error is returned, it can be due to errors in compilation of the match specification. Such errors are presented as a list of tuples `{error, string()}` where the string is a textual explanation of the compilation error. An example:

```
(x@y)4> dbg:tp({dbg,ltp,0},[{[],[],[{message, two, arguments}, {noexist}]}]).
{error,
 [{error,"Special form 'message' called with wrong number of
          arguments in {message,two,arguments}."},
  {error,"Function noexist/1 does_not_exist."}]}
```

`tpl(Module,MatchSpec)`

Same as tpl({Module, '_', '_'}, MatchSpec)

`tpl(Module,Function,MatchSpec)`

Same as tpl({Module, Function, '_'}, MatchSpec)

`tpl(Module, Function, Arity, MatchSpec)`

Same as tpl({Module, Function, Arity}, MatchSpec)

`tpl({Module, Function, Arity}, MatchSpec) -> {ok, MatchDesc} | {error, term()}`

This function works as `tp/2`, but enables tracing for loacl calls (and local functions) as well as for global calls (and functions).

`ctp(Module)`

Same as ctp({Module, '_', '_'})

`ctp(Module, Function)`

Same as ctp({Module, Function, '_'})

`ctp(Module, Function, Arity)`

Same as ctp({Module, Function, Arity})

`ctp({Module, Function, Arity}) -> {ok, MatchDesc} | {error, term()}`

Types:

- Module = atom() | '_'
- Function = atom() | '_'
- Arity = integer() | '_'
- MatchDesc = [MatchNum]
- MatchNum = {matched, integer()} | {matched, node(), integer()} | {matched, node(), 0, RPCError}

This function disables call tracing on the specified functions. The semantics of the parameter is the same as for the corresponding function specification in `tp/2` or `tpl/2`. Both local and global call trace is disabled.

The return value reflects how many functions that matched, and is constructed as described in `tp/2`. No tuple `{saved, N}` is however ever returned (for obvious reasons).

`ctpl(Module)`

Same as ctpl({Module, '_', '_'})

`ctpl(Module, Function)`

Same as ctpl({Module, Function, '_'})

`ctpl(Module, Function, Arity)`

Same as ctpl({Module, Function, Arity})

`ctpl({Module, Function, Arity}) -> {ok, MatchDesc} | {error, term()}`

This function works as `ctp/1`, but only disables tracing set up with `tpl/2` (not with `tp/2`).

`ctpg(Module)`

Same as ctpg({Module, '_', '_'})

`ctpg(Module, Function)`

Same as ctpg({Module, Function, '_'})

`ctpg(Module, Function, Arity)`

Same as ctpg({Module, Function, Arity})

`ctpg({Module, Function, Arity}) -> {ok, MatchDesc} | {error, term()}`

This function works as `ctp/1`, but only disables tracing set up with `tp/2` (not with `tpl/2`).

`ltp() -> ok`

Use this function to recall all match_spec's previously used in the session (i. e. previously saved during calls to `tp/2`. This is very useful, as a complicated match_spec can be quite awkward to write. Note that the match_spec's are lost if `stop/0` is called.

Match specifications used can be saved in a file (if a read-write file system is present) for use in later debugging sessions, see `wtp/1` and `rtp/1`

`dtp() -> ok`

Use this function to "forget" all match specifications saved during calls to `tp/2`. This is useful when one wants to restore other match specifications from a file with `rtp/1`. Use `dtp/1` to delete specific saved match specifications.

`dtp(N) -> ok`

Types:

- N = integer()

Use this function to "forget" a specific match specification saved during calls to `tp/2`.

`wtp(Name) -> ok | {error, IOError}`

Types:

- Name = string()
- IOError = term()

This function will save all match specifications saved during the session (during calls to `tp/2`) in a text file with the name designated by `Name`. The format of the file is textual, why it can be edited with an ordinary text editor, and then restored with `rtp/1`.

Each match spec in the file ends with a full stop (`.`) and new (syntactically correct) match specifications can be added to the file manually.

The function returns `ok` or an error tuple where the second element contains the I/O error that made the writing impossible.

`rtp(Name) -> ok | {error, Error}`

Types:

- Name = string()
- Error = term()

This function reads match specifications from a file (possibly) generated by the `wtp/1` function. It checks the syntax of all match specifications and verifies that they are correct. The error handling principle is "all or nothing", i. e. if some of the match specifications are wrong, none of the specifications are added to the list of saved match specifications for the running system.

The match specifications in the file are *merged* with the current match specifications, so that no duplicates are generated. Use `ltp/0` to see what numbers were assigned to the specifications from the file.

The function will return an error, either due to I/O problems (like a non existing or non readable file) or due to file format problems. The errors from a bad format file are in a more or less textual format, which will give a hint to what's causing the problem.

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

Types:

- Nodename = atom()
- Reason = term()

The `dbg` server keeps a list of nodes where tracing should be performed. Whenever a `tp/2` call or a `p/2` call is made, it is executed for all nodes in this list as well as the local node (except for `p/2` with a specific `pid()` as first argument, in which case the command is executed only on the node where the designated process resides.).

This function adds a node (`Nodename`) to the list of nodes where tracing is performed.

Distributed tracing does not work together with trace ports.

The function will return an error if either tracing is currently directed to a trace port (see `trace_port/2`) or the node `Nodename` is not reachable.

`cn(Nodename) -> ok`

Types:

- Nodename = atom()

Clears a node from the list of traced nodes. Subsequent calls to `tp/2` and `p/2` will not consider that node, but tracing already activated on the node will continue to be in effect.

Returns `ok`, cannot fail.

`ln() -> ok`

Shows the list of traced nodes on the console.

`tracer() -> {ok, pid()} | {error, already_started}`

This function starts a server that will be the recipient of all trace messages. All subsequent calls to `p/2` will result in messages sent to the newly started trace server.

A trace server started in this way will simply display the trace messages in a formatted way in the Erlang shell (i. e. use io:format). See `tracer/2` for a description of how the trace message handler can be customized.

`tracer(Type, Data) -> {ok, pid()} | {error, Error}`

Types:

- Type = port | process
- Data = PortGenerator | HandlerSpec
- HandlerSpec = {HandlerFun, InitialData}
- HandlerFun = fun() (two arguments)
- InitialData = term()
- PortGenerator = fun() (no arguments)
- Error = term()

This function starts a tracer server with additional parameters. The first parameter, the `Type`, indicates if trace messages should be handled by a receiving process (`process`) or by a tracer port (`port`). For a description about tracer ports see `trace_port/2`.

If `Type` is a process, a message handler function can be specified (`HandlerSpec`). The handler function, which should be a `fun` taking two arguments, will be called for each trace message, with the first argument containing the message as it is and the second argument containing the return value from the last invocation of the fun. The initial value of the second parameter is specified in the `InitialData` part of the `HandlerSpec`. The `HandlerFun` may chose any appropriate action to take when invoked, and can save a state for the next invocation by returning it.

If `Type` is a port, then the second parameter should be a *fun* which takes no arguments and returns a newly opened trace port when called. Such a *fun* is preferably generated by calling `trace_port/2`.

Note that most `dbg` functions start the server automatically. Call this function with the appropriate arguments *before* calling any other functions in the module. The server can be stopped with a call to `stop/0` if it has been started in the default form by mistake.

If an error is returned, it can either be due to a tracer server already running (`{error,already_started}`) or due to the `HandlerFun` throwing an exception.

`trace_port(Type, Parameters) -> fun()`

Types:

- Type = ip | file
- Parameters = Filename | WrapFilesSpec | IPPortSpec
- Filename = string() | [string()] | atom()
- WrapFilesSpec = {Filename, wrap, Suffix} | {Filename, wrap, Suffix, WrapSize} | {Filename, wrap, Suffix, WrapSize, WrapCnt}
- Suffix = string()

- WrapSize = integer() >= 0 | {time, WrapTime}
- WrapTime = integer() >= 1
- WrapCnt = integer() >= 1
- IpPortSpec = PortNumber | {PortNumber, QueSize}
- PortNumber = integer()
- QueSize = integer()

This function creates a trace port generating *fun*. The *fun* takes no arguments and returns a newly opened trace port. The return value from this function is suitable as a second parameter to tracer/2, i. e. `dbg:tracer(port, dbg:trace_port(ip, 4711))`.

A trace port is an Erlang port to a dynamically linked in driver that handles trace messages directly, without the overhead of sending them as messages in the Erlang virtual machine.

Two trace drivers are currently implemented, the `file` and the `ip` trace drivers. The file driver sends all trace messages into one or several binary files, from where they later can be fetched and processed with the `trace_client/2` function. The ip driver opens a TCP/IP port where it listens for connections. When a client (preferably started by calling `trace_client/2` on another Erlang node) connects, all trace messages are sent over the IP network for further processing by the remote client.

Using a trace port significantly lowers the overhead imposed by using tracing.

The file trace driver expects a filename or a wrap files specification as parameter. A file is written with a high degree of buffering, why all trace messages are *not* guaranteed to be saved in the file in case of a system crash. That is the price to pay for low tracing overhead.

A wrap files specification is used to limit the disk space consumed by the trace. The trace is written to a limited number of files each with a limited size. The actual filenames are `Filename ++ SeqCnt ++ Suffix`, where `SeqCnt` counts as a decimal string from `0` to `WrapCnt` and then around again from `0`. When a trace term written to the current file makes it longer than `WrapSize`, that file is closed, if the number of files in this wrap trace is as many as `WrapCnt` the oldest file is deleted then a new file is opened to become the current. Thus, when a wrap trace has been stopped, there are at most `WrapCnt` trace files saved with a size of at least `WrapSize` (but not much bigger), except for the last file that might even be empty. The default values are `WrapSize = 128*1024` and `WrapCnt = 8`.

The `SeqCnt` values in the filenames are all in the range `0` through `WrapCnt` with a gap in the circular sequence. The gap is needed to find the end of the trace.

If the `WrapSize` is specified as `{time, WrapTime}`, the current file is closed when it has been open more than `WrapTime` milliseconds, regardless of it being empty or not.

The ip trace driver has a queue of `QueSize` messages waiting to be delivered. If the driver cannot deliver messages as fast as they are produced by the runtime system, a special message is sent, which indicates how many messages that are dropped. That message will arrive at the handler function specified in `trace_client/3` as the tuple `{drop, N}` where `N` is the number of consecutive messages dropped. In case of heavy tracing, drop's are likely to occur, and they surely occur if no client is reading the trace messages.

Note that processes on other nodes cannot be traced using a trace port.

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

Equivalent to `trace_port_control(flush)`.

`trace_port_control(Operation) -> ok | {ok, Result}, {error, Reason}`

This function is used to do a control operation on the active trace port driver. Which operations that are allowed as well as their return values are depending on which trace driver that is used.

Returns either `ok` or `{ok, Result}` if the operation was successful, or `{error, Reason}` if the current tracer is a process or if it is a port not supporting the operation.

The allowed values for `Operation` are:

`flush` This function is used to flush the internal buffers held by a trace port driver. Currently only the file trace driver supports this operation. Returns `ok`.

`get_listen_port` Returns `{ok, IpPort}` where `IpPort`is the IP port number used by the driver listen socket. Only the ip trace driver supports this operation.

`trace_client(Type, Parameters) -> pid()`

Types:

- Type = ip | file | follow_file
- Parameters = Filename | WrapFilesSpec | IPClientPortSpec
- Filename = string() | [string()] | atom()
- WrapFilesSpec = see trace_port/2
- Suffix = string()
- IpClientPortSpec = PortNumber | {Hostname, PortNumber}
- PortNumber = integer()
- Hostname = string()

This function starts a trace client that reads the output created by a trace port driver and handles it in mostly the same way as a tracer process created by the `tracer/0` function.

If `Type` is `file`, the client reads all trace messages stored in the file named `Filename` or specified by `WrapFilesSpec` (must be the same as used when creating the trace, see trace_port/2) and let's the default handler function format the messages on the console. This is one way to interpret the data stored in a file by the file trace port driver.

If `Type` is `follow_file`, the client behaves as in the `file` case, but keeps trying to read (and process) more data from the file until stopped by `stop_trace_client/1`. `WrapFilesSpec` is not allowed as second argument for this `Type`.

If `Type` is `ip`, the client connects to the TCP/IP port `PortNumber` on the host `Hostname`, from where it reads trace messages until the TCP/IP connection is closed. If no `Hostname` is specified, the local host is assumed.

As an example, one can let trace messages be sent over the network to another Erlang node (preferably *not* distributed), where the formatting occurs:

On the node `stack` there's an Erlang node `ant@stack`, in the shell, type the following:

```
ant@stack> dbg:tracer(port, dbg:trace_port(ip,4711)).
<0.17.0>
ant@stack> dbg:p(self(), send).
{ok, 1}
```

All trace messages are now sent to the trace port driver, which in turn listens for connections on the TCP/IP port 4711. If we want to see the messages on another node, preferably on another host, we do like this:

```
-> dbg:trace_client(ip, {"stack", 4711}).
<0.42.0>
```

If we now send a message from the shell on the node `ant@stack`, where all sends from the shell are traced:

```
ant@stack> self() ! hello.
hello
```

The following will appear at the console on the node that started the trace client:

```
(<0.23.0>) <0.23.0> ! hello
(<0.23.0>) <0.22.0> ! {shell_rep,<0.23.0>,{value,hello,[],[]}}
```

The last line is generated due to internal message passing in the Erlang shell. The process id's will vary.

`trace_client(Type, Parameters, HandlerSpec) -> pid()`

Types:

- Type = ip | file | follow_file
- Parameters = Filename | WrapFilesSpec | IPClientPortSpec
- Filename = string() | [string()] | atom()
- WrapFilesSpec = see trace_port/2
- Suffix = string()
- IpClientPortSpec = PortNumber | {Hostname, PortNumber}
- PortNumber = integer()
- Hostname = string()
- HandlerSpec = {HandlerFun, InitialData}
- HandlerFun = fun() (two arguments)
- InitialData = term()

This function works exactly as `trace_client/2`, but allows you to write your own handler function. The handler function works mostly as the one described in `tracer/2`, but will also have to be prepared to handle trace messages of the form `{drop, N}`, where `N` is the number of dropped messages. This pseudo trace message will only occur if the ip trace driver is used.

For trace types `ip` and `file`, the pseudo trace message `end_of_trace` will appear at the end of the trace. The return value from the handler function is in this case ignored.

`stop_trace_client(Pid) -> ok`

Types:

- Pid = pid()

This function shuts down a previously started trace client. The `Pid` argument is the process id returned from the `trace_client/2` or `trace_client/3` call.

`get_tracer() -> {ok, Tracer}`

Types:

- Tracer = port() | pid()

Returns the process or port to which all trace messages are sent.

`stop() -> stopped`

Stops the dbg server and clears all trace flags for all processes. Also shuts down all trace clients and closes all trace ports.

Advanced topics - combining with seq_trace

The dbg module is primarily targeted towards tracing through the erlang:trace/3 function. It is sometimes desired to trace messages in a more delicate way, which can be done with the help of the seq_trace module.

Seq_trace implements sequential tracing (known in the AXE10 world, and sometimes called "forlopp tracing"). dbg can interpret messages generated from seq_trace and the same tracer function for both types of tracing can be used. The seq_trace messages can even be sent to a trace port for further analysis.

As a match specification can turn on sequential tracing, the combination of dbg and seq_trace can be quite powerful. This brief example shows a session where sequential tracing is used:

```
1> dbg:tracer().
{ok,<0.30.0>}
2> {ok, Tracer} = dbg:get_tracer().
{ok,<0.31.0>}
3> seq_trace:set_system_tracer(Tracer).
false
4> dbg:tp(dbg, get_tracer, [{[],[],[{set_seq_token, send, true}]}]).
{ok,[{matched,1},{saved,1}]}
5> dbg:p(all,call).
{ok,[{matched,22}]}
6> dbg:get_tracer(), seq_trace:set_token([]).
(<0.25.0>) call dbg:get_tracer()
SeqTrace [0]: (<0.25.0>) <0.30.0> ! {<0.25.0>,get_tracer} [Serial: {2,4}]
SeqTrace [0]: (<0.30.0>) <0.25.0> ! {dbg,{ok,<0.31.0>}} [Serial: {4,5}]
ok
```

This session sets the system_tracer to the same process as the ordinary tracer process (i. e. <0.31.0>) and sets the trace pattern for the function dbg:get_tracer to one that has the action of setting a sequential token. When the function is called by a traced process (all processes are traced in this case), the process gets "contaminated" by the token and seq_trace messages are sent both for the server request and the response. The seq_trace:set_token([]) after the call clears the seq_trace token, why no messages are sent when the answer propagates via the shell to the console port. The output would otherwise have been more noisy.

Index of Modules and Functions

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

c/3
  dbg , 6

c/4
  dbg , 6

cn/1
  dbg , 10

ctp/1
  dbg , 8

ctp/2
  dbg , 8

ctp/3
  dbg , 8

ctpg/1
  dbg , 9

ctpg/2
  dbg , 9

ctpg/3
  dbg , 9

ctpl/1
  dbg , 8

ctpl/2
  dbg , 9

ctpl/3
  dbg , 9

dbg
  c/3, 6
  c/4, 6
  cn/1, 10
  ctp/1, 8
  ctp/2, 8
  ctp/3, 8
  ctpg/1, 9
  ctpg/2, 9
  ctpg/3, 9
  ctpl/1, 8
  ctpl/2, 9
  ctpl/3, 9
  dtp/0, 9
  dtp/1, 9
  flush_trace_port/0, 12
  get_tracer/0, 14
  h/0, 5
  h/1, 5
  i/0, 7
  ln/0, 11
  ltp/0, 9
  n/1, 10
  p/1, 5
  p/2, 5
  rtp/1, 10
  stop/0, 15
  stop_trace_client/1, 14
  tp/2, 7
  tp/3, 7
  tp/4, 7
  tpl/2, 8
  tpl/3, 8
  tpl/4, 8
  trace_client/2, 13
  trace_client/3, 14
  trace_port/2, 11
  trace_port_control/1, 13
  tracer/0, 11
  tracer/2, 11
  wtp/1, 9

dtp/0
  dbg , 9

dtp/1
  dbg , 9

flush_trace_port/0
  dbg , 12

get_tracer/0

dbg , 14

h/0
dbg , 5

h/1
dbg , 5

i/0
dbg , 7

ln/0
dbg , 11

ltp/0
dbg , 9

n/1
dbg , 10

p/1
dbg , 5

p/2
dbg , 5

rtp/1
dbg , 10

stop/0
dbg , 15

stop_trace_client/1
dbg , 14

tp/2
dbg , 7

tp/3
dbg , 7

tp/4
dbg , 7

tpl/2
dbg , 8

tpl/3
dbg , 8

tpl/4
dbg , 8

trace_client/2
dbg , 13

trace_client/3
dbg , 14

trace_port/2
dbg , 11

trace_port_control/1
dbg , 13

tracer/0
dbg , 11

tracer/2
dbg , 11

wtp/1
dbg , 9