Runtime_Tools

Runtime_Tools 1.8.3
May 17 2010

May 17 2010

1 Reference Manual

Runtime_Tools provides low footprint tracing/debugging tools suitable for inclusion in a production system.

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` BIFs. 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.

For some examples of how to use `dbg` from the Erlang shell, see the *simple example* section.

The utilities are also 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* section.

Exports

`fun2ms(LiteralFun) -> MatchSpec`

Types:

LiteralFun = fun() literal

MatchSpec = term()

Pseudo function that by means of a `parse_transform` translates the *literal*`fun()` typed as parameter in the function call to a match specification as described in the `match_spec` manual of `ERTS` users guide. (with literal I mean that the `fun()` needs to textually be written as the parameter of the function, it cannot be held in a variable which in turn is passed to the function).

The parse transform is implemented in the module `ms_transform` and the source *must* include the file `ms_transform.hrl` in STDLIB for this pseudo function to work. Failing to include the hrl file in the source will result in a runtime error, not a compile time ditto. The include file is easiest included by adding the line `-include_lib("stdlib/include/ms_transform.hrl").` to the source file.

The `fun()` is very restricted, it can take only a single parameter (the parameter list to match), a sole variable or a list. It needs to use the `is_`XXX guard tests and one cannot use language constructs that have no representation in a match_spec (like `if`, `case`, `receive` etc). The return value from the fun will be the return value of the resulting match_spec.

Example:

```
1> dbg:fun2ms(fun([M,N]) when N > 3 -> return_trace() end).
[{['$1','$2'],[{'>','$2',3}],[{return_trace}]}]
```

Variables from the environment can be imported, so that this works:

```
2> X=3.
3
3> dbg:fun2ms(fun([M,N]) when N > X  -> return_trace() end).
[{['$1','$2'],[{'>','$2',{const,3}}],[{return_trace}]}]
```

The imported variables will be replaced by match_spec `const` expressions, which is consistent with the static scoping for Erlang `fun()`s. Local or global function calls can not be in the guard or body of the fun however. Calls to builtin match_spec functions of course is allowed:

```
4> dbg:fun2ms(fun([M,N]) when N > X, is_atomm(M)  -> return_trace() end).
Error: fun containing local erlang function calls ('is_atomm' called in guard) cannot be translated into mat
{error,transform_error}
5> dbg:fun2ms(fun([M,N]) when N > X, is_atom(M)  -> return_trace() end).
[{['$1','$2'],[{'>','$2',{const,3}},{is_atom,'$1'}],[{return_trace}]}]
```

As you can see by the example, the function can be called from the shell too. The `fun()` needs to be literally in the call when used from the shell as well. Other means than the parse_transform are used in the shell case, but more or less the same restrictions apply (the exception being records, as they are not handled by the shell).

> ### Warning:
>
> If the parse_transform is not applied to a module which calls this pseudo function, the call will fail in runtime (with a `badarg`). The module `dbg` actually exports a function with this name, but it should never really be called except for when using the function in the shell. If the `parse_transform` is properly applied by including the `ms_transform.hrl` header file, compiled code will never call the function, but the function call is replaced by a literal match_spec.

More information is provided by the `ms_transform` manual page in STDLIB.

`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, 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. The process may be a remote process (on another Erlang node). The node must be in the list of traced nodes (*see*`n/1` and `tracer/0/2/3`).
- 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` or `tracer/0/2/3` 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` or `tracer/0/2/3` 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` or `tracer/0/2/3` function.

- If the `Item` is an atom other than `all`, `new` or `existing`, the process with the corresponding registered name is traced.The process may be a remote process (on another Erlang node). The node must be added with the `n/1` or `tracer/0/2/3` function.
- If the `Item` is an integer, the process `<0.Item.0>` is traced.
- If the `Item` is a tuple `{X, Y, Z}`, the process `<X.Y.Z>` is traced.
- If the `Item` is a string "<X.Y.Z>" as returned from `pid_to_list/1`, 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 (procs)`

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 is `{matched, Node, N}`. 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, several or even all nodes failed.

`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() | atom() | [] | match_spec()

MatchDesc = [MatchInfo]

MatchInfo = {saved, integer()} | MatchNum

MatchNum = {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` or `tracer/0/2/3` 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. There are also built-in aliases named with atoms (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 local calls (and local functions) as well as for global calls (and functions).

ctp()

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

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, 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()

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

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()

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

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 specifications previously used in the session (i. e. previously saved during calls to `tp/2`, and built-in match specifications. This is very useful, as a complicated match_spec can be quite awkward to write. Note that the match specifications 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`) and built-in match specifications 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 including 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 remote node (`Nodename`) to the list of nodes where tracing is performed. It starts a tracer process on the remote node, which will send all trace messages to the tracer process on the local node (via the Erlang distribution). If no tracer process is running on the local node, the error reason `no_local_tracer` is returned. The tracer process on the local node must be started with the `tracer/0/2` function.

If `Nodename` is the local node, the error reason `cant_add_local_node` is returned.

If a trace port (*see*`trace_port/2`) is running on the local node, remote nodes can not be traced with a tracer process. The error reason `cant_trace_remote_pid_to_local_port` is returned. A trace port can however be started on the remote node with the `tracer/3` function.

The function will also return an error if 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 on the local node 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.

To start a similar tracer on a remote node, use `n/1`.

`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 on the local node. 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`.

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.

To start a similar tracer on a remote node, use `tracer/3`.

`tracer(Nodename, Type, Data) -> {ok, Nodename} | {error, Reason}`

Types:

Nodename = atom()

This function is equivalent to `tracer/2`, but acts on the given node. A tracer is started on the node (`Nodename`) and the node is added to the list of traced nodes.

> Note:
>
> This function is not equivalent to `n/1`. While `n/1` starts a process tracer which redirects all trace information to a process tracer on the local node (i.e. the trace control node), `tracer/3` starts a tracer of any type which is independent of the tracer on the trace control node.

For details, *see* `tracer/2`.

`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.

`flush_trace_port()`

Equivalent to `flush_trace_port(node())`.

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

Equivalent to `trace_port_control(Nodename,flush)`.

`trace_port_control(Operation)`

Equivalent to `trace_port_control(node(),Operation)`.

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

Types:

Nodename = atom()

This function is used to do a control operation on the active trace port driver on the given node (`Nodename`). 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 type `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()`

Equivalent to `get_tracer(node())`.

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

Types:

Nodename = atom()
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 and all trace patterns for all functions. Also shuts down all trace clients and closes all trace ports.

Note that no trace patterns are affected by this function.

`stop_clear() -> stopped`

Same as stop/0, but also clears all trace patterns on local and global functions calls.

Simple examples - tracing from the shell

The simplest way of tracing from the Erlang shell is to use `dbg:c/3` or `dbg:c/4`, e.g. tracing the function `dbg:get_tracer/0`:

```
(tiger@durin)84> dbg:c(dbg,get_tracer,[]).
(<0.154.0>) <0.152.0> ! {<0.154.0>,{get_tracer,tiger@durin}}
(<0.154.0>) out {dbg,req,1}
(<0.154.0>) << {dbg,{ok,<0.153.0>}}
(<0.154.0>) in {dbg,req,1}
(<0.154.0>) << timeout
{ok,<0.153.0>}
(tiger@durin)85>
```

Another way of tracing from the shell is to explicitly start a *tracer* and then set the *trace flags* of your choice on the processes you want to trace, e.g. trace messages and process events:

```
(tiger@durin)66> Pid = spawn(fun() -> receive {From,Msg} -> From ! Msg end end).
<0.126.0>
(tiger@durin)67> dbg:tracer().
{ok,<0.128.0>}
(tiger@durin)68> dbg:p(Pid,[m,procs]).
{ok,[{matched,tiger@durin,1}]}
(tiger@durin)69> Pid ! {self(),hello}.
(<0.126.0>) << {<0.116.0>,hello}
{<0.116.0>,hello}
(<0.126.0>) << timeout
(<0.126.0>) <0.116.0> ! hello
(<0.126.0>) exit normal
(tiger@durin)70> flush().
Shell got hello
ok
(tiger@durin)71>
```

If you set the `call` trace flag, you also have to set a *trace pattern* for the functions you want to trace:

```
(tiger@durin)77> dbg:tracer().
{ok,<0.142.0>}
(tiger@durin)78> dbg:p(all,call).
{ok,[{matched,tiger@durin,3}]}
(tiger@durin)79> dbg:tp(dbg,get_tracer,0,[]).
{ok,[{matched,tiger@durin,1}]}
(tiger@durin)80> dbg:get_tracer().
(<0.116.0>) call dbg:get_tracer()
{ok,<0.143.0>}
(tiger@durin)81> dbg:tp(dbg,get_tracer,0,[{'_',[],[{return_trace}]}]).
{ok,[{matched,tiger@durin,1},{saved,1}]}
(tiger@durin)82> dbg:get_tracer().
(<0.116.0>) call dbg:get_tracer()
(<0.116.0>) returned from dbg:get_tracer/0 -> {ok,<0.143.0>}
{ok,<0.143.0>}
(tiger@durin)83>
```

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, 0, [{[],[],[{set_seq_token, send, true}]}]).
{ok,[{matched,nonode@nohost,1},{saved,1}]}
5> dbg:p(all,call).
{ok,[{matched,nonode@nohost,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}]
{1,0,5,<0.30.0>,4}
```

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.

Note of caution

When tracing function calls on a group leader process (an IO process), there is risk of causing a deadlock. This will happen if a group leader process generates a trace message and the tracer process, by calling the trace handler function, sends an IO request to the same group leader. The problem can only occur if the trace handler prints to tty using an `io` function such as `format/2`. Note that when `dbg:p(all,call)` is called, IO processes are also traced. Here's an example:

```
%% Using a default line editing shell
1> dbg:tracer(process, {fun(Msg,_) -> io:format("~p~n", [Msg]), 0 end, 0}).
{ok,<0.37.0>}
2> dbg:p(all, [call]).
{ok,[{matched,nonode@nohost,25}]}
3> dbg:tp(mymod,[{'_',[],[]}]).
{ok,[{matched,nonode@nohost,0},{saved,1}]}
4> mymod: % TAB pressed here
%% -- Deadlock --
```

Here's another example:

```
%% Using a shell without line editing (oldshell)
1> dbg:tracer(process).
{ok,<0.31.0>}
2> dbg:p(all, [call]).
{ok,[{matched,nonode@nohost,25}]}
3> dbg:tp(lists,[{'_',[],[]}]).
{ok,[{matched,nonode@nohost,0},{saved,1}]}
% -- Deadlock --
```

The reason we get a deadlock in the first example is because when TAB is pressed to expand the function name, the group leader (which handles character input) calls `mymod:module_info()`. This generates a trace message which, in turn, causes the tracer process to send an IO request to the group leader (by calling `io:format/2`). We end up in a deadlock.

In the second example we use the default trace handler function. This handler prints to tty by sending IO requests to the `user` process. When Erlang is started in oldshell mode, the shell process will have `user` as its group leader and so will the tracer process in this example. Since `user` calls functions in `lists` we end up in a deadlock as soon as the first IO request is sent.

Here are a few suggestions for how to avoid deadlock:

- Don't trace the group leader of the tracer process. If tracing has been switched on for all processes, call `dbg:p(TracerGLPid,clear)` to stop tracing the group leader (`TracerGLPid`). `process_info(TracerPid,group_leader)` tells you which process this is (`TracerPid` is returned from `dbg:get_tracer/0`).
- Don't trace the `user` process if using the default trace handler function.
- In your own trace handler function, call `erlang:display/1` instead of an `io` function or, if `user` is not used as group leader, print to `user` instead of the default group leader. Example: `io:format(user,Str,Args)`.

erts_alloc_config

Erlang module

> **Note:**
>
> `erts_alloc_config` is currently an experimental tool and might be subject to backward incompatible changes.

erts_alloc(3) is an Erlang Run-Time System internal memory allocator library. `erts_alloc_config` is intended to be used to aid creation of an *erts_alloc(3)* configuration that is suitable for a limited number of runtime scenarios. The configuration that `erts_alloc_config` produce is intended as a suggestion, and may need to be adjusted manually.

The configuration is created based on information about a number of runtime scenarios. It is obviously impossible to foresee every runtime scenario that can occur. The important scenarios are those that cause maximum or minimum load on specific memory allocators. Load in this context is total size of memory blocks allocated.

The current implementation of `erts_alloc_config` concentrate on configuration of multi-block carriers. Information gathered when a runtime scenario is saved is mainly current and maximum use of multi-block carriers. If a parameter that change the use of multi-block carriers is changed, a previously generated configuration is invalid and `erts_alloc_config` needs to be run again. It is mainly the single block carrier threshold that effects the use of multi-block carriers, but other single-block carrier parameters might as well. If another value of a single block carrier parameter than the default is desired, use the desired value when running `erts_alloc_config`.

A configuration is created in the following way:

- Pass the *+Mea config* command-line flag to the Erlang runtime system you are going to use for creation of the allocator configuration. It will disable features that prevent `erts_alloc_config` from doing it's job. Note, you should *not* use this flag when using the created configuration. Also note that it is important that you use the same *amount of schedulers* when creating the configuration as you are going the use on the system using the configuration.
- Run your applications with different scenarios (the more the better) and save information about each scenario by calling *save_scenario/0*. It may be hard to know when the applications are at an (for `erts_alloc_config`) important runtime scenario. A good approach may therefore be to call *save_scenario/0* repeatedly, e.g. once every tenth second. Note that it is important that your applications reach the runtime scenarios that are important for `erts_alloc_config` when you are saving scenarios; otherwise, the configuration may perform bad.
- When you have covered all scenarios, call *make_config/1* in order to create a configuration. The configuration is written to a file that you have chosen. This configuration file can later be read by an Erlang runtime-system at startup. Pass the command line argument *-args_file FileName* to the *erl(1)* command.
- The configuration produced by `erts_alloc_config` may need to be manually adjusted as already stated. Do not modify the file produced by `erts_alloc_config`; instead, put your modifications in another file and load this file after the file produced by `erts_alloc_config`. That is, put the *-args_file FileName* argument that reads your modification file later on the command-line than the *-args_file FileName* argument that reads the configuration file produced by `erts_alloc_config`. If a memory allocation parameter appear multiple times, the last version of will be used, i.e., you can override parameters in the configuration file produced by `erts_alloc_config`. Doing it this way simplifies things when you want to rerun `erts_alloc_config`.

> **Note:**
>
> The configuration created by `erts_alloc_config` may perform bad, ever horrible, for runtime scenarios that are very different from the ones saved when creating the configuration. You are, therefore, advised to rerun `erts_alloc_config` if the applications run when the configuration was made are changed, or if the load on the applications have changed since the configuration was made. You are also advised to rerun `erts_alloc_config` if the Erlang runtime system used is changed.

`erts_alloc_config` saves information about runtime scenarios and performs computations in a server that is automatically started. The server register itself under the name `'__erts_alloc_config__'`.

Exports

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

Types:

Error = term()

`save_scenario/0` saves information about the current runtime scenario. This information will later be used when *make_config/0*, or *make_config/1* is called.

The first time `save_scenario/0` is called a server will be started. This server will save runtime scenarios. All saved scenarios can be removed by calling *stop/0*.

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

Types:

Error = term()

This is the same as calling *make_config(group_leader())*.

`make_config(FileNameOrIODev) -> ok | {error, Error}`

Types:

FileNameOrIODev = string() | io_device()

Error = term()

`make_config/1` uses the information previously saved by *save_scenario/0* in order to produce an `erts_alloc` configuration. At least one scenario have had to be saved. All scenarios previously saved will be used when creating the configuration.

If `FileNameOrIODev` is a `string()`, `make_config/1` will use `FileNameOrIODev` as a filename. A file named `FileNameOrIODev` is created and the configuration will be written to that file. If `FileNameOrIODev` is an *io_device()* (see the documentation of the module *io*), the configuration will be written to the io device.

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

Types:

Error = term()

Stops the server that saves runtime scenarios.

See Also

erts_alloc(3), *erl(1)*, *io(3)*