

Observer

Copyright © 2002-2026 Ericsson AB. All Rights Reserved.
Observer 2.15.1
November 2, 2026

Copyright © 2002-2026 Ericsson AB. All Rights Reserved.
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. Ericsson AB. All Rights Reserved..

November 2, 2026

1 Observer User's Guide

1.1 Introduction

1.1.1 Scope

The Observer application is a container including the following tools for tracing and investigation of distributed systems:

- Observer
- Trace Tool Builder
- Erlang Top
- Crashdump Viewer

1.1.2 Prerequisites

It is assumed that the reader is familiar with the Erlang programming language.

1.2 Observer

1.2.1 Introduction

Observer is a graphical tool for observing the characteristics of Erlang systems. Observer displays system information, application supervisor trees, process information, ETS tables, Mnesia tables and contains a front end for Erlang tracing.

1.2.2 Getting Started

Run Observer from a standalone node to minimize the impact of the system being observed.

Example:

```
% erl -sname observer -hidden -setcookie MyCookie -run observer
```

Select the node to observe with menu **Nodes**. Menu **View** > **Refresh interval** controls how often the view is to be updated. The refresh interval is set per viewer so you can have different settings for each viewer. To minimize the system impact, only the active viewer is updated. Other views are updated when activated.

The mouse buttons behave as expected. Use left-click to select objects, right-click to get a menu with the most used options, and double-click to display information about the selected object. In most viewers with many columns, you can change the sort order by left-clicking the column header.

1.2.3 System Tab

Tab **System** displays general information about the active Erlang node and its runtime system, such as build configuration, system capabilities, and overall use statistics.

1.2.4 Load Charts Tab

Tab **Load Charts** displays graphs of the current resource use on the active Erlang node.

Graph `Scheduler Utilization` shows scheduler use per scheduler, where each scheduler use has a unique color.

Graph `Memory Usage` shows the total memory use and per memory category use, where each category has a unique color. The categories are as follows:

`Total`

The sum of all memory categories.

`Processes`

The sum of all process memory used.

`Atom`

The size used by the atom table.

`Binary`

The sum of all off-heap binaries allocated.

`Code`

The memory allocated for code storage.

`Ets`

The used memory for all ETS tables.

Graph `IO Usage` shows the current I/O load on the system.

1.2.5 Memory Allocators Tab

Tab **Memory Allocators** displays detailed information of the carrier size and current memory carriers. For details about memory carriers, see module `erts_alloc` in application ERTS.

The `Max Carrier size` column shows the maximum value seen by observer since the last node change or since the start of the application, i.e. switching nodes will reset the max column. Values are sampled so higher values may have existed than what is shown.

1.2.6 Applications Tab

Tab **Applications** presents application information. Select an application in the left list to display its supervisor tree. The right-click options in the tree are as follows:

Process info

Opens a detailed information window on the selected process, including the following:

Process Information

Shows the process information.

Messages

Shows the process messages.

Dictionary

Shows the process dictionary.

Stack Trace

Shows the process current stack trace.

State

Shows the process state.

Log

If enabled and available, shows the process SASL log entries.

Trace process

Adds the selected process identifier to tab **Trace Overview** plus the node that the process resides on.

Trace named process

Adds the registered name of the process. This can be useful when tracing on many nodes, as processes with that name are then traced on all traced nodes.

Trace process tree

Adds the selected process and all processes below, right of it, to tab **Trace Overview**.

Trace named process tree

Adds the selected process and all processes below, right of it, to tab **Trace Overview**.

1.2.7 Processes Tab

Tab **Processes** lists process information in columns. For each process the following information is displayed:

Pid

The process identifier.

Reds

The number of reductions executed on the process. This can be presented as accumulated values or as values since the last update.

Memory

The size of the process, in bytes, obtained by a call to `process_info(Pid,memory)`.

MsgQ

The length of the message queue for the process.

Option **Process info** opens a detailed information window on the process under the mouse pointer, including the following:

Process Information

Shows the process information.

Messages

Shows the process messages.

Dictionary

Shows the process dictionary.

Stack Trace

Shows the process current stack trace.

State

Shows the process state.

Log

If enabled and available, shows the process SASL log entries.

> **Note:**
>
> **Log** requires application SASL to be started on the observed node, with `log_mf_h` as log handler. The Observed node must be Erlang/OTP R16B02 or higher. The `rb` server must not be started on the observed node when clicking menu **Log** > **Toggle log view**. The `rb` server is stopped on the observed node when exiting or changing the observed node.

Option **Trace selected processes** adds the selected process identifiers to tab **Trace Overview** plus the node that the processes reside on.

Option **Trace selected processes by name** adds the registered name of the processes. This can be useful when tracing is done on many nodes, as processes with that name are then traced on all traced nodes.

Option **Kill process** brutally kills the processes under the mouse pointer by sending an exit signal with reason `kill`.

1.2.8 Ports Tab

Tab **Ports** lists port information in columns. For each port the following information is displayed:

Id

The port identifier.

Connected

The process identifier for the process that owns the port.

Name

The registered name of the port, if any.

Controls

The name of the command set by `erlang:open_port/2`.

Slot

The internal index of the port.

Option **Port info** opens a detailed information window for the port under the mouse pointer. In addition to the information above, it also shows links and monitors.

Option **Trace selected ports** adds the selected port identifiers, and the nodes that the ports reside on, to tab **Trace Overview**.

Option **Trace selected ports by name** adds the registered name of the port to tab **Trace Overview**. This can be useful when tracing is done on many nodes, as ports with that name are then traced on all traced nodes.

Option **Close** executes `erlang:port_close/1` on the port under the mouse pointer.

1.2.9 Sockets Tab

Tab **Sockets** is divided into two parts. The first part contains general `socket` information and the second part lists socket information in columns.

For each socket the following information is displayed:

Id

The socket identifier.

Owner

The process identifier for the process that owns the socket.

Fd

The underlying file descriptor of the socket.

Domain

The communication domain (e.g. inet or inet6) of this socket.

Type

The type (e.g. stream or dgram) of this socket.

Protocol

The protocol (e.g. tcp or udp) of this socket.

Read State

The read state of the socket.

Write State

The write state of the socket.

Option **Socket info** opens a detailed information window for the socket under the mouse pointer. In addition to the information above, it also shows monitors.

Option **Close** executes `socket:close/1` on the socket under the mouse pointer.

1.2.10 Table Viewer Tab

Tab **Table Viewer** lists tables. By default, ETS tables are displayed whereas unreadable private ETS tables and tables created by OTP applications are not displayed. Use menu **View** to view "system" ETS tables, unreadable ETS tables, or Mnesia tables.

Double-click to view the table content, or right-click and select option **Show Table Content**. To view table information, select the table and activate menu **View > Table information**, or right-click and select option **Table info**.

You can use regular expressions and search for objects, and edit or delete them.

1.2.11 Trace Overview Tab

Tab **Trace Overview** handles tracing. Trace by selecting the processes or ports to be traced and how to trace them. For processes, you can trace messages, function calls, scheduling, garbage collections, and process-related events such as `spawn`, `exit`, and many others. For ports, you can trace messages, scheduling and port-related events.

To trace function calls, you also need to set up **trace patterns**. Trace patterns select the function calls to be traced. The number of traced function calls can be further reduced with **match specifications**. Match specifications can also be used to trigger more information in the trace messages.

You can also set match specifications on messages. By default, if tracing messages, all messages sent and/or received by the process or port are traced. Match specifications can be used to reduce the number of traced messages and/or to trigger more information in the trace messages.

Note:

Trace patterns only apply to the traced processes and ports.

Processes are added from the **Applications** or **Processes** tabs. Ports are added from the **Ports** tab. A special **new** identifier, meaning all processes, or ports, started after trace start, can be added with buttons **Add 'new' Processes** and **Add 'new' Ports**, respectively.

When adding processes or ports, a window with trace options is displayed. The chosen options are set for the selected processes/ports. To change the options, right-click the process or port and select **Edit process options**. To remove a process or port from the list, right-click and select **Remove process** or **Remove port**, respectively.

Processes and ports added by process/port identifiers add the nodes these processes/ports reside on in the node list. More nodes can be added by clicking button **Add Nodes**, or by right-clicking in the **Nodes** list and select **Add Nodes**. To remove nodes, select them, then right-click and choose **Remove nodes**.

If function calls are traced, trace patterns must be added by clicking button **Add Trace Pattern**. Select a module, function(s), and a match specification. If no functions are selected, all functions in the module are traced.

Trace patterns can also be added for traced messages. Click button **Add Trace Pattern** and select **Messages sent** or **Messages received**, and a match specification.

A few basic match specifications are provided in the tool, and you can provide your own match specifications. The syntax of match specifications is described in the `ERTS User's Guide`. To simplify the writing of a match specification, they can also be written as `fun/1`. For details, see module ms_transform in application STDLIB.

Click button **Start Trace** to start the trace. By default, trace output is written to a new window. Tracing is stopped when the window is closed, or when clicking button **Stop Trace**. Trace output can be changed with menu **Options** > **Output**. The trace settings, including match specifications, can be saved to, or loaded from, a file.

For details about tracing, see module dbg in application Runtime_Tools and in section "Match specifications in Erlang" in `ERTS User's Guide` and in module `ms_transform` in application STDLIB.

1.3 Trace Tool Builder

1.3.1 Introduction

Trace Tool Builder is a base for building trace tools for single node or distributed Erlang systems. It requires the Runtime_Tools application to be available on the traced node.

The following are the main features of Trace Tool Builder:

- Start tracing to file ports on many nodes with one function call.
- Write more information to a trace information file, which is read during formatting.
- Restore previous configuration by maintaining a history buffer and handling configuration files.
- Provide some simple support for sequential tracing.
- Format binary trace logs and merge logs from multiple nodes.

The intention of Trace Tool Builder is to serve as a base for tailor-made trace tools, but it can also be used directly from the Erlang shell (it can mimic `dbg` behaviour while still providing useful additions, such as match specification shortcuts). Trace Tool Builder only allows the use of file port tracer, so to use other types of trace clients it is better to use `dbg` directly.

1.3.2 Getting Started

Module `ttb` is the interface to all functions in Trace Tool Builder.

To get started, the least you need to do is to start a tracer with `ttb:tracer/0,1,2`, and set the required trace flags on the processes you want to trace with `ttb:p/2`.

When the tracing is completed, stop the tracer with `ttb:stop/0,1` and format the trace log with `ttb:format/1,2` (if there is anything to format).

Useful functions:

`ttb:tracer/0,1,2`

Opens a trace port on each node to be traced. By default, trace messages are written to binary files on remote nodes (the binary trace log).

`ttb:p/2`

Specifies the processes to be traced. Trace flags specified in this call specify what to trace on each process. This function can be called many times if you like different trace flags to be set on different processes.

`ttb:tp/2,3,4` or `ttb:tpl/2,3,4`

If you want to trace function calls (that is, if you have trace flag `call` set on any process), you must also set trace patterns on the required function(s) with `ttb:tp/2,3,4` or `ttb:tpl/2,3,4`. A function is only traced if it has a trace pattern. The trace pattern specifies how to trace the function by using match specifications. Match specifications are described in the ERTS User's Guide.

`ttb:stop/0,1`

Stops tracing on all nodes, deletes all trace patterns, and flushes the trace port buffer.

`ttb:format/1/2`

Translates the binary trace logs into something readable. By default, `ttb` presents each trace message as a line of text, but you can also write your own handler to make more complex interpretations of the trace information. A trace log can also be presented graphically with application Event Tracer (ET).

If option `format` is specified to `ttb:stop/1`, the formatting is automatically done when stopping `ttb`.

Tracing Local Node from Erlang Shell

The following small module is used in the subsequent example:

```
-module(m).
-export([f/0]).
f() ->
   receive
      From when is_pid(From) ->
         Now = erlang:now(),
         From ! {self(),Now}
   end.
```

The following example shows the basic use of `ttb` from the Erlang shell. Default options are used both for starting the tracer and for formatting (the custom fetch directory is however provided). This gives a trace log named `Node-ttb` in the newly created directory, where `Node` is the node name. The default handler prints the formatted trace messages in the shell:

```
(tiger@durin)47> %% First I spawn a process running my test function
(tiger@durin)47> Pid = spawn(m,f,[]).
<0.125.0>
(tiger@durin)48>
(tiger@durin)48> %% Then I start a tracer...
(tiger@durin)48> ttb:tracer().
{ok,[tiger@durin]}
(tiger@durin)49>
(tiger@durin)49> %% and activate the new process for tracing
(tiger@durin)49> %% function calls and sent messages.
(tiger@durin)49> ttb:p(Pid,[call,send]).
{ok,[{<0.125.0>,[{matched,tiger@durin,1}]}]}
(tiger@durin)50>
(tiger@durin)50> %% Here I set a trace pattern on erlang:now/0
(tiger@durin)50> %% The trace pattern is a simple match spec
(tiger@durin)50> %% indicating that the return value should be
(tiger@durin)50> %% traced. Refer to the reference_manual for
(tiger@durin)50> %% the full list of match spec shortcuts
(tiger@durin)50> %% available.
(tiger@durin)51> ttb:tp(erlang,now,return).
{ok,[{matched,tiger@durin,1},{saved,1}]}
(tiger@durin)52>
(tiger@durin)52> %% I run my test (i.e. send a message to
(tiger@durin)52> %% my new process)
(tiger@durin)52> Pid ! self().
<0.72.0>
(tiger@durin)53>
(tiger@durin)53> %% And then I have to stop ttb in order to flush
(tiger@durin)53> %% the trace port buffer
(tiger@durin)53> ttb:stop([return, {fetch_dir, "fetch"}]).
{stopped, "fetch"}
(tiger@durin)54>
(tiger@durin)54> %% Finally I format my trace log
(tiger@durin)54> ttb:format("fetch").
({<0.125.0>,{m,f,0},tiger@durin}) call erlang:now()
({<0.125.0>,{m,f,0},tiger@durin}) returned from erlang:now/0 ->
{1031,133451,667611}
({<0.125.0>,{m,f,0},tiger@durin}) <0.72.0> !
{<0.125.0>,{1031,133451,667611}}
ok
```

Build Your Own Tool

The following example shows a simple tool for "debug tracing", that is, tracing of function calls with return values:

```
-module(mydebug).
-export([start/0,trc/1,stop/0,format/1]).
-export([print/4]).
%% Include ms_transform.hrl so that I can use dbg:fun2ms/2 to
%% generate match specifications.
-include_lib("stdlib/include/ms_transform.hrl").
%%% ------------Tool API------------
%%% --------------------------------
%%% Star the "mydebug" tool
start() ->
    %% The options specify that the binary log shall be named
    %% <Node>-debug_log and that the print/4 function in this
    %% module shall be used as format handler
    ttb:tracer(all,[{file,"debug_log"},{handler,{{?MODULE,print},0}}]),
    %% All processes (existing and new) shall trace function calls
    %% We want trace messages to be sorted upon format, which requires
    %% timestamp flag. The flag is however enabled by default in ttb.
    ttb:p(all,call).

%%% Set trace pattern on function(s)
trc(M) when is_atom(M) ->
    trc({M,'_','_'});
trc({M,F}) when is_atom(M), is_atom(F) ->
    trc({M,F,'_'});
trc({M,F,_A}=MFA) when is_atom(M), is_atom(F) ->
    %% This match spec shortcut specifies that return values shall
    %% be traced.
    MatchSpec = dbg:fun2ms(fun(_) -> return_trace() end),
    ttb:tpl(MFA,MatchSpec).

%%% Format a binary trace log
format(Dir) ->
    ttb:format(Dir).

%%% Stop the "mydebug" tool
stop() ->
    ttb:stop(return).

%%% --------Internal functions--------
%%% --------------------------------
%%% Format handler
print(_Out,end_of_trace,_TI,N) ->
    N;
print(Out,Trace,_TI,N) ->
    do_print(Out,Trace,N),
    N+1.

do_print(Out,{trace_ts,P,call,{M,F,A},Ts},N) ->
    io:format(Out,
              "~w: ~w, ~w:~n"
              "Call      : ~w:~w/~w~n"
              "Arguments :~p~n~n",
              [N,Ts,P,M,F,length(A),A]);
do_print(Out,{trace_ts,P,return_from,{M,F,A},R,Ts},N) ->
    io:format(Out,
              "~w: ~w, ~w:~n"
              "Return from  : ~w:~w/~w~n"
              "Return value :~p~n~n",
              [N,Ts,P,M,F,A,R]).
```

To distinguish trace logs produced with this tool from other logs, option `file` is used in `tracer/2`. The logs are therefore fetched to a directory named `ttb_upload_debug_log-YYYYMMDD-HHMMSS`

By using option `handler` when starting the tracer, the information about how to format the file is stored in the trace information file (`.ti`). This is not necessary, as it can be specified when formatting instead. However, It can be useful if you, for example, want to format trace logs automatically using option `format` in `ttb:stop/1`. Also, you do not need any knowledge of the content of a binary log to format it the way it is intended. If option `handler` is specified both when starting the tracer and when formatting, the one specified when formatting is used.

Trace flag `call` is set on all processes. This means that any function activated with command `trc/1` is traced on all existing and new processes.

1.3.3 Running Trace Tool Builder against Remote Node

The Observer application might not always be available on the node to be traced (in the following called the "traced node"). However, Trace Tool Builder can still be run from another node (in the following called the "trace control node") as long as the following is fulfilled:

- The Observer application is available on the trace control node.
- The Runtime_Tools application is available on both the trace control node and the traced node.

If Trace Tool Builder is to be used against a remote node, it is highly recommended to start the trace control node as **hidden**. This way it can connect to the traced node without being "seen" by it, that is, if the `nodes()` BIF is called on the traced node, the trace control node does not show. To start a hidden node, add option `-hidden` to the `erl` command, for example:

```
% erl -sname trace_control -hidden
```

Diskless Node

If the traced node is diskless, `ttb` must be started from a trace control node with disk access, and option `file` must be specified to function `tracer/2` with value `{local, File}`, for example:

```
(trace_control@durin)1> ttb:tracer(mynode@diskless,
                                   {file,{local,{wrap,"mytrace"}}}).
{ok,[mynode@diskless]}
```

1.3.4 More Tracing Options

When setting up a trace, the following features can also be activated:

- Time-constrained tracing
- Overload protection
- Autoresume
- `dbg` mode

Time-Constrained Tracing

It can sometimes be helpful to enable trace for a specified period of time (for example, to monitor a system for 24 hours or half a second). This can be done with option `{timer, TimerSpec}`. If `TimerSpec` has the form of `MSec`, the trace is stopped after `MSec` milliseconds using `ttb:stop/0`. If more options are provided (`TimerSpec = {MSec, Opts}`), `ttb:stop/1` is called instead with `Opts` as argument.

The timer is started with `ttb:p/2`, so any trace patterns must be set up in advance. `ttb:start_trace/4` always sets up all patterns before invoking `ttb:p/2`.

The following example shows how to set up a trace that is automatically stopped and formatted after 5 seconds:

```
(tiger@durin)1> ttb:start_trace([node()],
                                [{erlang, now,[]}],
                                {all, call},
                                [{timer, {5000, format}}]).
```

Note:

Because of network and processing delays, the period of tracing is approximate.

Overload Protection

When tracing live systems, always take special care to not overload a node with too heavy tracing. `ttb` provides option `overload` to address this problem.

`{overload, MSec, Module, Function}` instructs the `ttb` back end (a part of the Runtime_Tools application) to perform overload check every `MSec` millisecond. If the check (named `Module:Function(check)`) returns `true`, tracing is disabled on the selected node.

Overload protection activated on one node does not affect other nodes, where the tracing continues as normal. `ttb:stop/0,1` fetches data from all clients, including everything collected before the activation of overload protection.

Note:

It is not allowed to change trace details (with `ttb:p` and `ttb:tp/tpl...`) once overload protection is activated in one of the traced nodes. This is to avoid trace setup being inconsistent between nodes.

`Module:Function` provided with option `overload` must handle three calls: `init`, `check`, and `stop`. `init` and `stop` allow some setup and teardown required by the check. An overload check module can look as follows:

```
-module(overload).
-export([check/1]).

check(init) ->
    Pid = sophisticated_module:start(),
    put(pid, Pid);
check(check) ->
    get(pid) ! is_overloaded,
    receive
        Reply ->
            Reply
    after 5000 ->
            true
    end;
check(stop) ->
    get(pid) ! stop.
```

Note:

`check` is always called by the same process, so `put` and `get` are possible.

Autoresume

A node can crash (probably a buggy one, hence traced). Use `resume` to resume tracing on the node automatically when it gets back. The failing node then tries to reconnect to trace control node when `Runtime_Tools` is started.

This implies that `Runtime_Tools` must be included in the startup chain of other nodes (if not, you can still resume tracing by starting `Runtime_Tools` manually, that is, by an RPC call).

To not lose the data that the failing node stored up to the point of crash, the control node tries to fetch it before restarting trace. This must occur within the allowed time frame, otherwise it is aborted (default is 10 seconds, but it can be changed with `{resume, MSec}`). The data fetched this way is then merged with all other traces.

The autostart feature requires more data to be stored on traced nodes. By default, the data is stored automatically to the file named "ttb_autostart.bin" in the current working directory (cwd) of the traced node. Users can change this behaviour (that is, on diskless nodes) by specifying their own module to handle autostart data storage and retrieval (`ttb_autostart_module` environment variable of `runtime_tools`). For information about the API, see module `ttb`. The following example shows the default handler:

```
-module(ttb_autostart).
-export([read_config/0,
         write_config/1,
         delete_config/0]).

-define(AUTOSTART_FILENAME, "ttb_autostart.bin").

delete_config() ->
    file:delete(?AUTOSTART_FILENAME).

read_config() ->
    case file:read_file(?AUTOSTART_FILENAME) of
        {ok, Data} -> {ok, binary_to_term(Data)};
        Error      -> Error
    end.

write_config(Data) ->
    file:write_file(?AUTOSTART_FILENAME, term_to_binary(Data)).
```

Note:

Remember that file trace ports buffer the data by default. If the node crashes, trace messages are not flushed to the binary log. If the risk of failure is high, it can be a good idea to flush the buffers every now and then automatically. Passing `{flush, MSec}` as an option of `ttb:tracer/2` flushes all buffers every `MSec` millisecond.

dbg Mode

Option `{shell, ShellType}` allows making `ttb` operation similar to `dbg`. Using `{shell, true}` displays all trace messages in the shell before storing them. `{shell, only}` additionally disables message storage (making the tool to behave exactly like `dbg`). This is allowed only with IP trace ports (`{trace, {local, File}}`).

Command `ttb:tracer(dbg)` is a shortcut for the pure `dbg` mode (`{shell, only}`).

1.3.5 Trace Information and File .ti

In addition to the trace log file(s), a file with extension `.ti` is created when Trace Tool Builder is started. This is the trace information file. It is a binary file, which contains the process information, trace flags used, the name of the node to which it belongs, and all information written with function `ttb:write_trace_info/2`. `.ti` files are always fetched with other logs when the trace is stopped.

Except for the process information, everything in the trace information file is passed on to the handler function when formatting. Parameter `TI` is a list of `{Key,ValueList}` tuples. The keys `flags`, `handler`, `file`, and `node` are used for information written directly by `ttb`.

Information to the trace information file by can be added by calling `ttb:write_trace_info/2`. Notice that `ValueList` always is a list, and if you call `write_trace_info/2` many times with the same `Key`, the `ValueList` is extended with a new value each time.

Example:

`ttb:write_trace_info(mykey,1)` gives the entry `{mykey,[1]}` in `TI`. Another call, `ttb:write_trace_info(mykey,2)`, changes this entry to `{mykey,[1,2]}`.

1.3.6 Wrap Logs

If you want to limit the size of the trace logs, you can use wrap logs. This works almost like a circular buffer. You can specify the maximum number of binary logs and the maximum size of each log. `ttb` then creates a new binary log each time a log reaches the maximum size. When the maximum number of logs are reached, the oldest log is deleted before a new one is created.

Note:

The overall size of data generated by `ttb` can be greater than the wrap specification suggests. If a traced node restarts and autoresume is enabled, the old wrap log is always stored and a new one is created.

Wrap logs can be formatted one by one or all at once. See Formatting.

1.3.7 Formatting

Formatting can be done automatically when stopping `ttb` (see section Automatically Collect and Format Logs from All Nodes), or explicitly by calling function `ttb:format/1,2`.

Formatting means to read a binary log and present it in a readable format. You can use the default format handler in `ttb` to present each trace message as a line of text, or write your own handler to make more complex interpretations of the trace information. You can also use application ET to present the trace log graphically (see section Presenting Trace Logs with Event Tracer).

The first argument to `ttb:format/1,2` specifies which binary log(s) to format. This is usually the name of a directory that `ttb` created during log fetch. Unless option `disable_sort` is provided, the logs from different files are always sorted according to time-stamp in traces.

The second argument to `ttb:format/2` is a list of options as follows:

`out`

Specifies the destination to write the formatted text. Default destination is `standard_io`, but a filename can also be specified.

`handler`

Specifies the format handler to use. If this option is not specified, option `handler` that is specified when starting the tracer is used. If option `handler` is not specified when starting the tracer either, a default handler is used, which prints each trace message as a text line.

`disable_sort`

Indicates that the logs are not to be merged according to time-stamp, but processed one file after another (this can be a bit faster).

A format handler is a fun taking four arguments. This fun is called for each trace message in the binary log(s). A simple example that only prints each trace message can be as follows:

```
fun(Fd, Trace, _TraceInfo, State) ->
   io:format(Fd, "Trace: ~p~n", [Trace]),
   State
end.
```

Here, `Fd` is the file descriptor for the destination file, or the atom `standard_io`. `_TraceInfo` contains information from the trace information file (see section Trace Information and File .ti). `State` is a state variable for the format handler fun. The initial value of variable `State` is specified with the handler option, for example:

```
ttb:format("tiger@durin-ttb", [{handler, {{Mod,Fun}, initial_state}}])
                                                    ^^^^^^^^^^^^^
```

Another format handler can be used to calculate the time spent by the garbage collector:

```
fun(_Fd,{trace_ts,P,gc_start,_Info,StartTs},_TraceInfo,State) ->
      [{P,StartTs}|State];
   (Fd,{trace_ts,P,gc_end,_Info,EndTs},_TraceInfo,State) ->
      {value,{P,StartTs}} = lists:keysearch(P,1,State),
      Time = diff(StartTs,EndTs),
      io:format("GC in process ~w: ~w milliseconds~n", [P,Time]),
      State -- [{P,StartTs}]
end
```

A more refined version of this format handler is function `handle_gc/4` in module `multitrace.erl` included in directory `src` of the Observer application.

The trace message is passed as the second argument (`Trace`). The possible values of `Trace` are the following:

- All trace messages described in `erlang:trace/3`
- `{drop, N}` if IP tracer is used (see `dbg:trace_port/2`)
- `end_of_trace` received once when all trace messages are processed

By giving the format handler `ttb:get_et_handler()`, you can have the trace log presented graphically with `et_viewer` in the ET application (see section Presenting Trace Logs with Event Tracer).

You can always decide not to format the whole trace data contained in the fetch directory, but analyze single files instead. To do so, a single file (or list of files) must be passed as the first argument to `format/1,2`.

Wrap logs can be formatted one by one or all at once. To format one of the wrap logs in a set, specify the exact file name. To format the whole set of wrap logs, specify the name with `*` instead of the wrap count.

Example:

Start tracing:

```
(tiger@durin)1> ttb:tracer(node(),{file,{wrap,"trace"}}).
{ok,[tiger@durin]}
(tiger@durin)2> ttb:p(...)
...
```

This gives a set of binary logs, for example:

```
tiger@durin-trace.0.wrp
tiger@durin-trace.1.wrp
tiger@durin-trace.2.wrp
...
```

Format the whole set of logs:

```
1> ttb:format("tiger@durin-trace.*.wrp").
....
ok
2>
```

Format only the first log:

```
1> ttb:format("tiger@durin-trace.0.wrp").
....
ok
2>
```

To merge all wrap logs from two nodes:

```
1> ttb:format(["tiger@durin-trace.*.wrp","lion@durin-trace.*.wrp"]).
....
ok
2>
```

Presenting Trace Logs with Event Tracer

For detailed information about the Event Tracer, see the ET application.

By giving the format handler `ttb:get_et_handler()`, you can have the trace log presented graphically with `et_viewer` in the ET application. `ttb` provides filters that can be selected from the menu **Filter** in the `et_viewer` window. The filters are names according to the type of actors they present (that is, what each vertical line in the sequence diagram represents). Interaction between actors is shown as red arrows between two vertical lines, and activities within an actor are shown as blue text to the right of the actors line.

The `processes` filter is the only filter showing all trace messages from a trace log. Each vertical line in the sequence diagram represents a process. Erlang messages, spawn, and link/unlink are typical interactions between processes. Function calls, scheduling, and garbage collection, are typical activities within a process. `processes` is the default filter.

The remaining filters only show function calls and function returns. All other trace message are discarded. To get the most out of these filters, `et_viewer` must know the caller of each function and the time of return. This can be obtained using both the `call` and `return_to` flags when tracing. Notice that flag `return_to` only works with local call trace, that is, when trace patterns are set with `ttb:tpl`.

The same result can be obtained by using the flag `call` only and setting a match specification on local or global function calls as follows:

```
1> dbg:fun2ms(fun(_) -> return_trace(),message(caller()) end).
[{'_',[],[{return_trace},{message,{caller}}]}]
```

This must however be done with care, as function `{return_trace}` in the match specification destroys tail recursiveness.

The `modules` filter shows each module as a vertical line in the sequence diagram. External function calls/returns are shown as interactions between modules, and internal function calls/returns are shown as activities within a module.

The `functions` filter shows each function as a vertical line in the sequence diagram. A function calling itself is shown as an activity within a function, and all other function calls are shown as interactions between functions.

The `mods_and_procs` and `funcs_and_procs` filters are equivalent to the `modules` and `functions` filters respectively, except that each module or function can have many vertical lines, one for each process it resides on.

In the following example, modules `foo` and `bar` are used:

```
-module(foo).
-export([start/0,go/0]).

start() ->
    spawn(?MODULE, go, []).

go() ->
    receive
        stop ->
            ok;
        go ->
            bar:f1(),
            go()
    end.
```

```
-module(bar).
-export([f1/0,f3/0]).
f1() ->
    f2(),
    ok.
f2() ->
    spawn(?MODULE,f3,[]).
f3() ->
    ok.
```

Setting up the trace:

```
(tiger@durin)1> %%First we retrieve the Pid to limit traced processes set
(tiger@durin)1> Pid = foo:start().
(tiger@durin)2> %%Now we set up tracing
(tiger@durin)2> ttb:tracer().
(tiger@durin)3> ttb:p(Pid, [call, return_to, procs, set_on_spawn]).
(tiger@durin)4> ttb:tpl(bar, []).
(tiger@durin)5> %%Invoke our test function and see output with et viewer
(tiger@durin)5> Pid ! go.
(tiger@durin)6> ttb:stop({format, {handler, ttb:get_et_handler()}}).
```

This renders a result similar to the following:

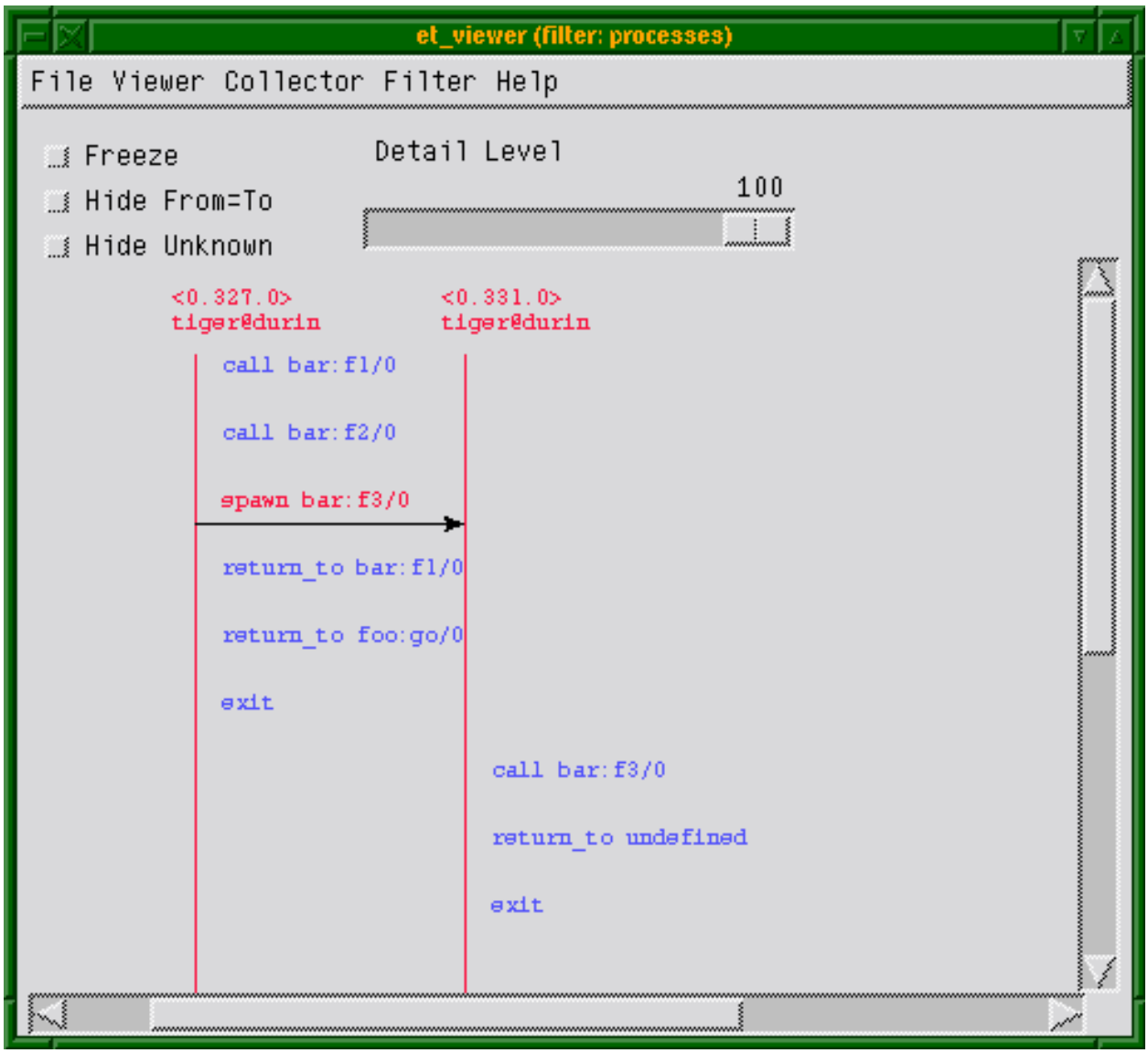


Figure 3.1: Filter: "processes"

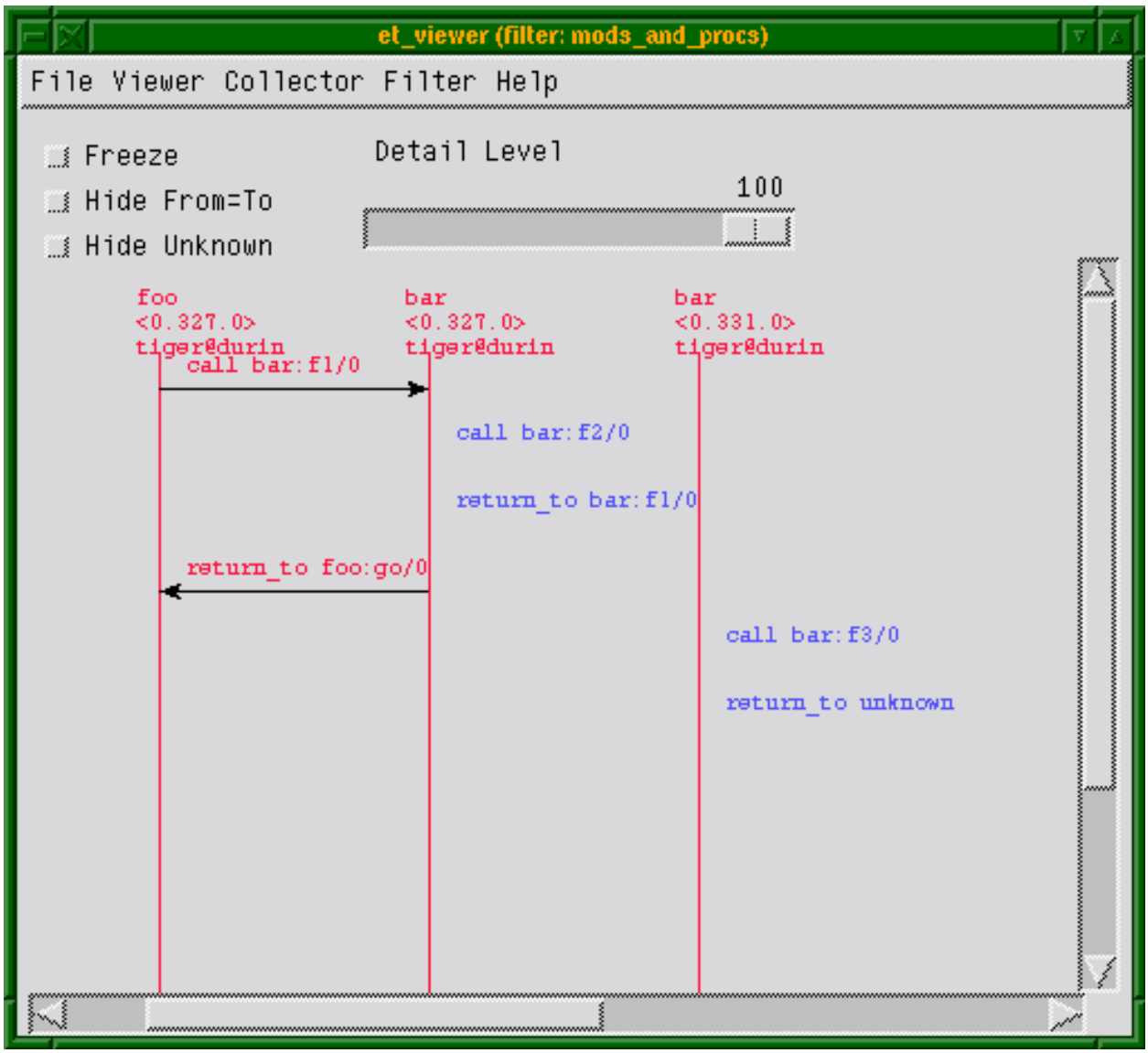


Figure 3.2: Filter: "mods_and_procs"

Notice that function `ttb:start_trace/4` can be used as help as follows:

```
(tiger@durin)1> Pid = foo:start().
(tiger@durin)2> ttb:start_trace([node()],
                                [{bar,[]}],
                                {Pid, [call, return_to, procs, set_on_spawn]}
                                {handler, ttb:get_et_handler()}).
(tiger@durin)3> Pid ! go.
(tiger@durin)4> ttb:stop(format).
```

1.3.8 Automatically Collect and Format Logs from All Nodes

By default, `ttb:stop/1` fetches trace logs and trace information files from all nodes. The logs are stored in a new directory named `ttb_upload-Filename-Timestamp` under the working directory of the trace control node.

Fetching can be disabled by providing option `nofetch` to `ttb:stop/1`. The user can specify a fetch directory by passing option `{fetch_dir, Dir}`.

If option `format` is specified to `ttb:stop/1`, the trace logs are automatically formatted after tracing is stopped.

1.3.9 History and Configuration Files

For the tracing functionality, `dbg` can be used instead of `ttb` for setting trace flags on processes and trace patterns for call trace, that is, the functions `p`, `tp`, `tpl`, `ctp`, `ctpl`, and `ctpg`. Only the following two things are added by `ttb` for these functions:

- All calls are stored in the history buffer and can be recalled and stored in a configuration file. This makes it easy to set up the same trace environment, for example, if you want to compare two test runs. It also reduces the amount of typing when using `ttb` from the Erlang shell.
- Shortcuts are provided for the most common match specifications (to not force you to use `dbg:fun2ms` continually).

Use `ttb:list_history/0` to see the content of the history buffer and `ttb:run_history/1` to re-execute one of the entries.

The main purpose of the history buffer is the possibility to create configuration files. Any function stored in the history buffer can be written to a configuration file and used for creating a specific configuration at any time with a single function call.

A configuration file is created or extended with `ttb:write_config/2,3`. Configuration files are binary files and can therefore only be read and written with functions provided by `ttb`.

The complete content of the history buffer can be written to a configuration file by calling `ttb:write_config(ConfigFile,all)`. Selected entries from the history can be written by calling `ttb:write_config(ConfigFile,NumList)`, where `NumList` is a list of integers pointing out the history entries to write. Moreover, the history buffer is always dumped to `ttb_last_config` when `ttb:stop/0,1` is called.

User-defined entries can also be written to a configuration file by calling function `ttb:write_config(ConfigFile,ConfigList)`, where `ConfigList` is a list of `{Module,Function,Args}`.

Any existing file `ConfigFile` is deleted and a new file is created when `write_config/2` is called. Option `append` can be used to add something at the end of an existing configuration file, for example, `ttb:write_config(ConfigFile,What,[append])`.

Example:

See the content of the history buffer:

```
(tiger@durin)191> ttb:tracer().
{ok,[tiger@durin]}
(tiger@durin)192> ttb:p(self(),[garbage_collection,call]).
{ok,{[<0.1244.0>],[garbage_collection,call]}}
(tiger@durin)193> ttb:tp(ets,new,2,[]).
{ok,[{matched,1}]}
(tiger@durin)194> ttb:list_history().
[{1,{ttb,tracer,[tiger@durin,[]]}},
 {2,{ttb,p,[<0.1244.0>,[garbage_collection,call]]}},
 {3,{ttb,tp,[ets,new,2,[]]}}]
```

Execute an entry from the history buffer:

```
(tiger@durin)195> ttb:ctp(ets,new,2).
{ok,[{matched,1}]}
(tiger@durin)196> ttb:list_history().
[{1,{ttb,tracer,[tiger@durin,[]]}},
 {2,{ttb,p,[<0.1244.0>,[garbage_collection,call]]}},
 {3,{ttb,tp,[ets,new,2,[]]}},
 {4,{ttb,ctp,[ets,new,2]}}]
(tiger@durin)197> ttb:run_history(3).
ttb:tp(ets,new,2,[]) ->
{ok,[{matched,1}]}
```

Write the content of the history buffer to a configuration file:

```
(tiger@durin)198> ttb:write_config("myconfig",all).
ok
(tiger@durin)199> ttb:list_config("myconfig").
[{1,{ttb,tracer,[tiger@durin,[]]}},
 {2,{ttb,p,[<0.1244.0>,[garbage_collection,call]]}},
 {3,{ttb,tp,[ets,new,2,[]]}},
 {4,{ttb,ctp,[ets,new,2]}},
 {5,{ttb,tp,[ets,new,2,[]]}}]
```

Extend an existing configuration:

```
(tiger@durin)200> ttb:write_config("myconfig",[{ttb,tp,[ets,delete,1,[]]}],
[append]).
ok
(tiger@durin)201> ttb:list_config("myconfig").
[{1,{ttb,tracer,[tiger@durin,[]]}},
 {2,{ttb,p,[<0.1244.0>,[garbage_collection,call]]}},
 {3,{ttb,tp,[ets,new,2,[]]}},
 {4,{ttb,ctp,[ets,new,2]}},
 {5,{ttb,tp,[ets,new,2,[]]}},
 {6,{ttb,tp,[ets,delete,1,[]]}}]
```

Go back to a previous configuration after stopping Trace Tool Builder:

```
(tiger@durin)202> ttb:stop().
ok
(tiger@durin)203> ttb:run_config("myconfig").
ttb:tracer(tiger@durin,[]) ->
{ok,[tiger@durin]}

ttb:p(<0.1244.0>,[garbage_collection,call]) ->
{ok,{[<0.1244.0>],[garbage_collection,call]}}

ttb:tp(ets,new,2,[]) ->
{ok,[{matched,1}]}

ttb:ctp(ets,new,2) ->
{ok,[{matched,1}]}

ttb:tp(ets,new,2,[]) ->
{ok,[{matched,1}]}

ttb:tp(ets,delete,1,[]) ->
{ok,[{matched,1}]}

ok
```

Write selected entries from the history buffer to a configuration file:

```
(tiger@durin)204> ttb:list_history().
[{1,{ttb,tracer,[tiger@durin,[]]}},
 {2,{ttb,p,[<0.1244.0>,[garbage_collection,call]]}},
 {3,{ttb,tp,[ets,new,2,[]]}},
 {4,{ttb,ctp,[ets,new,2]}},
 {5,{ttb,tp,[ets,new,2,[]]}},
 {6,{ttb,tp,[ets,delete,1,[]]}}]
(tiger@durin)205> ttb:write_config("myconfig",[1,2,3,6]).
ok
(tiger@durin)206> ttb:list_config("myconfig").
[{1,{ttb,tracer,[tiger@durin,[]]}},
 {2,{ttb,p,[<0.1244.0>,[garbage_collection,call]]}},
 {3,{ttb,tp,[ets,new,2,[]]}},
 {4,{ttb,tp,[ets,delete,1,[]]}}]
(tiger@durin)207>
```

1.3.10 Sequential Tracing

To learn what sequential tracing is and how it can be used, see the Reference Manual for `seq_trace`.

The support for sequential tracing provided by Trace Tool Builder includes the following:

- Initiation of the system tracer. This is automatically done when a trace port is started with `ttb:tracer/0,1,2`.
- Creation of match specifications that activates sequential tracing.

Starting sequential tracing requires that a tracer is started with function `ttb:tracer/0,1,2`. Sequential tracing can then be started in either of the following ways:

- Through a trigger function with a match specification created with `ttb:seq_trigger_ms/0,1`.
- Directly by using module `seq_trace`.

Example 1:

In the following example, function `dbg:get_tracer/0` is used as trigger for sequential tracing:

```
(tiger@durin)110> ttb:tracer().
{ok,[tiger@durin]}
(tiger@durin)111> ttb:p(self(),call).
{ok,{[<0.158.0>],[call]}}
(tiger@durin)112> ttb:tp(dbg,get_tracer,0,ttb:seq_trigger_ms(send)).
{ok,[{matched,1},{saved,1}]}
(tiger@durin)113> dbg:get_tracer(), seq_trace:reset_trace().
true
(tiger@durin)114> ttb:stop(format).
({<0.158.0>,{shell,evaluator,3},tiger@durin}) call dbg:get_tracer()
SeqTrace [0]: ({<0.158.0>,{shell,evaluator,3},tiger@durin})
{<0.237.0>,dbg,tiger@durin} ! {<0.158.0>,{get_tracer,tiger@durin}}
[Serial: {0,1}]
SeqTrace [0]: ({<0.237.0>,dbg,tiger@durin})
{<0.158.0>,{shell,evaluator,3},tiger@durin} ! {dbg,{ok,#Port<0.222>}}
[Serial: {1,2}]
ok
(tiger@durin)116>
```

Example 2:

Starting sequential tracing with a trigger is more useful if the trigger function is not called directly from the shell, but rather implicitly within a larger system. When calling a function from the shell, it is simpler to start sequential tracing directly, for example, as follows:

```
(tiger@durin)116> ttb:tracer().
{ok,[tiger@durin]}
(tiger@durin)117> seq_trace:set_token(send,true), dbg:get_tracer(),
seq_trace:reset_trace().
true
(tiger@durin)118> ttb:stop(format).
SeqTrace [0]: ({<0.158.0>,{shell,evaluator,3},tiger@durin})
{<0.246.0>,dbg,tiger@durin} ! {<0.158.0>,{get_tracer,tiger@durin}}
[Serial: {0,1}]
SeqTrace [0]: ({<0.246.0>,dbg,tiger@durin})
{<0.158.0>,{shell,evaluator,3},tiger@durin} ! {dbg,{ok,#Port<0.229>}}
[Serial: {1,2}]
ok
(tiger@durin)120>
```

In both previous examples, `seq_trace:reset_trace/0` resets the trace token immediately after the traced function to avoid many trace messages because of the printouts in the Erlang shell.

All functions in module `seq_trace`, except `set_system_tracer/1`, can be used after the trace port is started with `ttb:tracer/0,1,2`.

1.3.11 Multipurpose Trace Tool

Module `multitrace` in directory `src` of the Observer application provides a small tool with three possible trace settings. The trace messages are written to binary files, which can be formatted with function `multitrace:format/1,2`:

`multitrace:debug(What)`

Start calltrace on all processes and trace the specified function(s). The format handler used is `multitrace:handle_debug/4` that prints each call and returns. `What` must be an item or a list of items to trace, specified on the format `{Module,Function,Arity}`, `{Module,Function}`, or only `Module`.

`multitrace:gc(Procs)`

Trace garbage collection on the specified process(es). The format handler used is `multitrace:handle_gc/4` that prints start, stop, and the time spent for each garbage collection.

`multitrace:schedule(Procs)`

Trace in-scheduling and out-scheduling on the specified process(es). The format handler used is `multitrace:handle_schedule/4` that prints each in-scheduling and out-scheduling with process, time-stamp, and current function. It also prints the total time each traced process was scheduled in.

1.4 Erlang Top

1.4.1 Introduction

Erlang Top, `etop`, is a tool for presenting information about Erlang processes similar to the information presented by `top` in UNIX.

1.4.2 Getting Started

Start Erlang Top in either of the following ways:

- Use script `etop`.
- Use batch file `etop.bat`, for example, `etop -node tiger@durin`.

1.4.3 Output

The output from Erlang Top is as follows:

```
================================================================================
 tiger@durin                                                          13:40:32
 Load:  cpu         0               Memory:  total        1997    binary         33
        procs     197                        processes       0    code          173
        runq      135                        atom         1002    ets            95

Pid            Name or Initial Func    Time    Reds  Memory    MsgQ Current Function
--------------------------------------------------------------------------------
<127.23.0>     code_server                0   59585   78064       0 gen_server:loop/6
<127.21.0>     file_server_2              0   36380   44276       0 gen_server:loop/6
<127.2.0>      erl_prim_loader            0   27962    3740       0 erl_prim_loader:loop
<127.9.0>      kernel_sup                 0    6998    4676       0 gen_server:loop/6
<127.17.0>     net_kernel                62    6018    3136       0 gen_server:loop/6
<127.0.0>      init                       0    4156    4352       0 init:loop/1
<127.16.0>     auth                       0    1765    1264       0 gen_server:loop/6
<127.18.0>     inet_tcp_dist:accept       0     660    1416       0 prim_inet:accept0/2
<127.5.0>      application_controll       0     569    6756       0 gen_server:loop/6
<127.137.0>    net_kernel:do_spawn_       0     553    5840       0 dbg:do_relay_1/1
================================================================================
```

The header includes some system information:

`Load`

`cpu`

`Runtime/Wallclock`, that is, the percentage of time where the node has been active.

`procs`

The number of processes on the node.

`runq`

The number of processes that are ready to run.

`Memory`

The memory allocated by the node in kilobytes.

For each process the following information is presented:

`Time`

The runtime for the process, that is, the time that the process has been scheduled in.

`Reds`

The number of reductions executed on the process.

`Memory`

The size of the process in bytes, obtained by a call to `process_info(Pid,memory)`.

`MsgQ`

The length of the message queue for the process.

> **Note:**
>
> **Time** and **Reds** can be presented as accumulated values or as values since the last update.

1.4.4 Configuration

All configuration parameters can be set at start by adding `-OptName Value` to the command line, for example:

```
% etop -node tiger@durin -setcookie mycookie -lines 15
```

A list of all valid Erlang Top configuration parameters is available in module `etop`.

The parameters `lines`, `interval`, `accumulate`, and `sort` can be changed during runtime with function `etop:config/2`.

Example:

Change configuration parameter `lines` with text-based presentation. Before the change, 10 lines are presented as follows:

```
============================================================================================
 tiger@durin                                                              10:12:39
 Load:  cpu         0               Memory:  total        1858    binary         33
        procs     191                        processes       0    code          173
        runq        2                        atom         1002    ets            95

Pid            Name or Initial Func    Time    Reds  Memory    MsgQ Current Function
--------------------------------------------------------------------------------------------
<127.23.0>     code_server                0   60350   71176       0 gen_server:loop/6
<127.21.0>     file_server_2              0   36380   44276       0 gen_server:loop/6
<127.2.0>      erl_prim_loader            0   27962    3740       0 erl_prim_loader:loop
<127.17.0>     net_kernel                 0   13808    3916       0 gen_server:loop/6
<127.9.0>      kernel_sup                 0    6998    4676       0 gen_server:loop/6
<127.0.0>      init                       0    4156    4352       0 init:loop/1
<127.18.0>     inet_tcp_dist:accept       0    2196    1416       0 prim_inet:accept0/2
<127.16.0>     auth                       0    1893    1264       0 gen_server:loop/6
<127.43.0>     ddll_server                0     582    3744       0 gen_server:loop/6
<127.5.0>      application_controll       0     569    6756       0 gen_server:loop/6
============================================================================================
```

Function `etop:config/2` is called to change the number of showed lines to 5:

```
> etop:config(lines,5).
ok
```

After the change, 5 lines are presented as follows:

```
(etop@durin)2>
================================================================================
 tiger@durin                                                          10:12:44
 Load:  cpu         0              Memory:  total        1859    binary         33
        procs     192                       processes       0    code          173
        runq        2                       atom         1002    ets            95

Pid            Name or Initial Func    Time    Reds  Memory    MsgQ Current Function
--------------------------------------------------------------------------------
<127.17.0>     net_kernel               183      70    4092       0 gen_server:loop/6
<127.335.0>    inet_tcp_dist:do_acc     141      22    1856       0 dist_util:con_loop/9
<127.19.0>     net_kernel:ticker/2      155       6    1244       0 net_kernel:ticker1/2
<127.341.0>    net_kernel:do_spawn_       0       0    5840       0 dbg:do_relay_1/1
<127.43.0>     ddll_server                0       0    3744       0 gen_server:loop/6
================================================================================
```

1.4.5 Print to File

At any time, the current Erlang Top display can be dumped to a text file with function `etop:dump/1`.

1.4.6 Stop

To stop Erlang Top, use function `etop:stop/0`.

1.5 Crashdump Viewer

1.5.1 Introduction

The Crashdump Viewer is a WxWidgets based tool for browsing Erlang crashdumps.

1.5.2 Getting Started

The easiest way to start Crashdump Viewer is to use shell script `cdv` with the full path to the Erlang crashdump as argument. The script is located in directory `priv` of the Observer application. This starts the Crashdump Viewer GUI and loads the specified file. If no filename is specified, a file dialog is opened where the file can be selected.

Under Windows, the batch file `cdv.bat` can be used.

Crashdump Viewer can also be started from an Erlang node by calling crashdump_viewer:start/0 or crashdump_viewer:start/1.

1.5.3 GUI

The GUI main window is opened when Crashdump Viewer has loaded a crashdump. It contains a title bar, a menu bar, information tabs, and a status bar.

The title bar shows the name of the currently loaded crashdump.

The menu bar contains a **File** menu and a **Help** menu. From the **File** menu, a new crashdump can be loaded or the tool can be terminated. From the **Help** menu, this User's Guide and section "How to interpret the Erlang crash dumps" from the ERTS application can be opened. "How to interpret the Erlang crash dumps" describes the raw crashdumps in detail and includes information about each field in the information pages."How to interpret the Erlang crash dumps" is also available in the OTP online documentation.

The status bar at the bottom of the window shows a warning if the currently loaded dump is truncated.

The center area of the main window contains the information tabs. Each tab displays information about a specific item or a list of items. Select a tab by clicking the tab title.

From tabs displaying lists of items, for example, the **Processes** tab or the **Ports** tab, a new window with more information can be opened by double-clicking a row or by right- clicking the row and selecting an item from the drop-down menu. The new window is called a detail window. Detail windows can be opened for processes, ports, nodes, and modules.

The information shown in a detail window can contain links to processes or ports. Clicking one of these links opens the detail window for the process or port in question. If the process or port resides on a remote node, no information is available. Clicking the link then displays a dialog where you can choose to open the detail window for the remote node.

Some tabs contain a left-hand menu where subitems of the information area can be selected. Click one of the rows, and the information is displayed in the right-hand information area.

1.5.4 Tab Content

Each tab in the main window contains an information page. If no information is found for an item, the page is empty. The reason for not finding information about an item can be the following:

- It is a dump from an old OTP release in which this item was not written.
- The item was not present in the system at the point of failure.
- The dump is truncated. In this case, a warning is displayed in the status bar of the main window.

Even if some information about an item exists, there can be empty fields if the dump originates from an old OTP release.

The value `-1` in any field means "unknown", and in most cases it means that the dump was truncated somewhere around this field.

The following sections describe some of the fields in the information tabs. These are fields that do not exist in the raw crashdump, or in some way differ from the fields in the raw crashdump. For details about other fields, see the ERTS User's Guide, section "How to interpret the Erlang crash dumps". That section can also be opened from the **Help** menu in the main window. There are also links from the following sections to related information in "How to interpret the Erlang crash dumps".

1.5.5 General Tab

Tab **General** shows a short overview of the dump.

The following fields are not described in the ERTS User's Guide:

`Crashdump created on`

Time of failure.

`Memory allocated`

The total number of bytes allocated, equivalent to `c:memory(total)`.

`Memory maximum`

The maximum number of bytes that has been allocated during the lifetime of the originating node. This is only shown if the Erlang runtime system is run instrumented.

`Atoms`

If available in the dump, this is the total number of atoms in the atom table. If the size of the atom table is unavailable, the number of atoms visible in the dump is displayed.

`Processes`

The number of processes visible in the dump.

`ETS tables`

The number of ETS tables visible in the dump.

`Funs`

The number of funs visible in the dump.

For details, see General Information in section "How to Interpret the Erlang Crash Dumps" in ERTS.

1.5.6 Processes Tab

Tab **Processes** shows a list of all processes found in the crashdump, including brief information about each process. By default, the processes are sorted by their pids. To sort by another topic, click the desired column heading.

Column **Memory** shows the 'Memory' field that was added to crashdumps in Erlang/OTP R16B01. This is the total amount of memory used by the process. For crashdumps from earlier releases, this column shows the 'Stack+heap' field. The value is always in bytes.

To view detailed information about a specific process, double- click the row in the list, or right-click the row and select **Properties for <pid>**.

For details, see Process Information in section "How to Interpret the Erlang Crash Dumps" in ERTS.

1.5.7 Ports Tab

Tab **Ports** is similar to the **Processes** tab, except it lists all ports found in the crashdump.

To view more details about a specific port, double-click the row or right-click it and select **Properties for <port>**. From the right-click menu, you can also select **Properties for <pid>**, where `<pid>` is the process connected to the port.

For details, see Port Information in section "How to Interpret the Erlang Crash Dumps" in ERTS.

1.5.8 ETS Tables Tab

Tab **ETS Tables** shows all ETS table information found in the dump. **Id** is the same as the 'Table' field in the raw crashdump. **Memory** is the 'Words' field from the raw crashdump translated into bytes. For tree tables, there is no value in the 'Objects' field.

To open the detailed information page about the table, double- click, or right-click the row and select **Properties for '<Identifier>'**.

To open the detailed information page about the owner process of an ETS table, right-click the row and select **Properties for <pid>**.

For details, see ETS Tables in section "How to Interpret the Erlang Crash Dumps" in ERTS.

1.5.9 Timers Tab

Tab **Timers** shows all timer information found in the dump.

To open the detailed information page about the owner process of a timer, right-click the row and select **Properties for <pid>**.

Double-clicking a row in the **Timers** tab has no effect.

For details, see Timers in section "How to Interpret the Erlang Crash Dumps" in ERTS.

1.5.10 Schedulers Tab

Tab **Schedulers** shows all scheduler information found in the dump.

To open the detailed information page about the scheduler, double-click, or right-click the row and select **Properties for '<Identifier>'**.

For details, see Scheduler Information in section "How to Interpret the Erlang Crash Dumps" in ERTS.

1.5.11 Funs Tab

Tab **Funs** shows all fun information found in the dump.

To open the detailed information page about the module to which the fun belongs, right-click the row and select **Properties for <mod>**.

Double-clicking a row in the **Funs** tab has no effect.

For details, see Fun Information in section "How to Interpret the Erlang Crash Dumps" in ERTS.

1.5.12 Atoms Tab

Tab **Atoms** lists all atoms found in the dump. By default the atoms are sorted in creation order from first to last. This is opposite of the raw crashdump where atoms are listed from last to first, meaning that if the dump was truncated in the middle of the atom list, only the last created atoms are visible in the **Atoms** tab.

For details, see Atoms in section "How to Interpret the Erlang Crash Dumps" in ERTS.

1.5.13 Nodes Tab

Tab **Nodes** shows a list of all external Erlang nodes that are referenced from the crashdump.

If the page is empty, it means either of the following:

- The crashed node is not distributed.
- The crashed node is distributed but has no references to other nodes.
- The dump is truncated.

If the node is distributed, all referenced nodes are visible. Column **Connection type** shows if the node is visible, hidden, or not connected. Visible nodes are alive nodes with a living connection to the originating node. Hidden nodes are the same as visible nodes, except they are started with flag `-hidden`. Not connected nodes are nodes that are not connected to the originating node anymore, but references (that is, process or port identifiers) exist.

To see more detailed information about a node, double-click the row, or right-click the row and select **Properties for node <node>**. From the right-click menu, you can also select **Properties for <port>**, to open the detailed information window for the controlling port.

In the detailed information window for a node, any existing links and monitors between processes on the originating node and the connected node are displayed. **Extra Info** can contain debug information (that is, special information written if the emulator is debug-compiled) or error information.

For details, see Distribution Information in section "How to Interpret the Erlang Crash Dumps" in ERTS.

1.5.14 Modules Tab

Tab **Modules** lists all modules loaded on the originating node, and the current code size. If old code exists, the old size is also shown.

To view detailed information about a specific module, double- click the row, or right-click it and select **Properties for <mod>**.

For details, see Loaded Module Information in section "How to Interpret the Erlang Crash Dumps" in ERTS.

1.5.15 Memory Tab

Tab **Memory** shows memory and allocator information. From the left-hand menu you can select the following:

Memory

See Memory Information in section "How to Interpret the Erlang Crash Dumps" in ERTS.

Allocator Summary

This page presents a summary of values from all allocators underneath it.

<Allocator>

One entry per allocator. See Allocator in section "How to Interpret the Erlang Crash Dumps" in ERTS.

Allocated Areas

See Allocated Areas in section "How to Interpret the Erlang Crash Dumps" in ERTS.

1.5.16 Internal Tables Tab

On tab **Internal Tables** you can from the left-hand menu select **Hash Tables**, **Index Tables**, or **Internal ETS Tables**. For details, see Internal Table Information in section "How to Interpret the Erlang Crash Dumps" in ERTS.

2 Reference Manual

Observer

Application

The Observer application contains tools for tracing and investigation of distributed systems.

Configuration

No configuration parameters are available for this application.

observer

Erlang module

Observer is a graphical tool for observing the characteristics of Erlang systems. The tool Observer displays system information, application supervisor trees, process information, ETS tables, Mnesia tables, and contains a front end for Erlang tracing with module `ttb`.

For details about how to get started, see the `User's Guide`.

Exports

`start() -> ok`

Starts the Observer GUI. To stop the tool, close the window or call stop/0.

`start(Node) -> ok`

Starts the Observer GUI and tries to connect it to `Node`.

`start_and_wait() -> ok`

Starts the Observer GUI and only return when it is either stopped or the window is closed

`start_and_wait(Node) -> ok`

Starts the Observer GUI and only return when it is either stopped or the window is closed, connects it directly to `Node` like start/1.

`stop() -> ok`

Stops the Observer GUI.

ttb

Erlang module

The Trace Tool Builder, `ttb`, is a base for building trace tools for distributed systems.

When using `ttb`, do not use module `dbg` in application Runtime_Tools in parallel.

Exports

`start_trace(Nodes, Patterns, FlagSpec, Opts) -> Result`

Types:

```
Result = see p/2
Nodes = see tracer/2
Patterns = [tuple()]
FlagSpec = {Procs, Flags}
Proc = see p/2
Flags = see p/2
Opts = see tracer/2
```

This function is a shortcut allowing to start a trace with one command. Each tuple in `Patterns` is converted to a list, which in turn is passed to `ttb:tpl/2,3,4`.

The call:

```
> ttb:start_trace([Node, OtherNode],
                  [{mod, foo, []}, {mod, bar, 2}],
                  {all, call},
                  [{file, File}, {handler,{fun myhandler/4, S}}]).
```

is equivalent to:

```
> ttb:start_trace([Node, OtherNode],
                  [{file, File}, {handler,{fun myhandler/4, S}}]),
ttb:tpl(mod, foo, []),
ttb:tpl(mod, bar, 2, []),
ttb:p(all, call).
```

`tracer() -> Result`

Equivalent to `tracer(node())`.

`tracer(Shortcut) -> Result`

Types:

```
Shortcut = shell | dbg
```

Handy shortcuts for common tracing settings.

`shell` is equivalent to `tracer(node(),[{file, {local, "ttb"}}, shell])`.

`dbg` is equivalent to `tracer(node(),[{shell, only}])`.

tracer(Nodes) -> Result

Equivalent to `tracer(Nodes,[])`.

tracer(Nodes,Opts) -> Result

Types:

```
Result = {ok, ActivatedNodes} | {error,Reason}
Nodes = atom() | [atom()] | all | existing | new
Opts = Opt | [Opt]
Opt = {file,Client} | {handler, FormatHandler} | {process_info,PI} |
shell | {shell, ShellSpec} | {timer, TimerSpec} | {overload_check, {MSec,
Module, Function}} | {flush, MSec} | resume | {resume, FetchTimeout} |
{queue_size, QueueSize}
TimerSpec = MSec | {MSec, StopOpts}
MSec = FetchTimeout = integer()
Module = Function = atom()
StopOpts = see stop/2
Client = File | {local, File}
File = Filename | Wrap
Filename = string()
Wrap = {wrap,Filename} | {wrap,Filename,Size,Count}
FormatHandler = See format/2
PI = true | false
ShellSpec = true | false | only
QueueSize = non_neg_integer()
```

Starts a file trace port on all specified nodes and points the system tracer for sequential tracing to the same port.

Options:

`Filename`

The specified `Filename` is prefixed with the node name. Default `Filename` is `ttb`.

`File={wrap,Filename,Size,Count}`

Can be used if the size of the trace logs must be limited. Default values are `Size=128*1024` and `Count=8`.

`Client`

When tracing diskless nodes, `ttb` must be started from an external "trace control node" with disk access, and `Client` must be `{local, File}`. All trace information is then sent to the trace control node where it is written to file.

`queue_size`

When tracing to shell or `{local,File}`, an ip trace driver is used internally. The ip trace driver has a queue of maximum `QueueSize` messages waiting to be delivered. If the driver cannot deliver messages as fast as they are produced, the queue size might be exceeded and messages are dropped. This parameter is optional, and is only useful if many `{drop,N}` trace messages are received by the trace handler. It has no meaning if shell or `{local,File}` is not used. See dbg:trace_port/2 for more information about the ip trace driver.

`process_info`

Indicates if process information is to be collected. If `PI = true` (which is default), each process identifier `Pid` is replaced by a tuple `{Pid,ProcessInfo,Node}`, where `ProcessInfo` is the registered process name, its globally registered name, or its initial function. To turn off this functionality, set `PI = false`.

`{shell, ShellSpec}`

Indicates that trace messages are to be printed on the console as they are received by the tracing process. This implies trace client `{local, File}`. If `ShellSpec` is `only` (instead of `true`), no trace logs are stored.

`shell`

Shortcut for `{shell, true}`.

`timer`

Indicates that the trace is to be automatically stopped after `MSec` milliseconds. `StopOpts` are passed to command `ttb:stop/2` if specified (default is `[]`). Notice that the timing is approximate, as delays related to network communication are always present. The timer starts after `ttb:p/2` is issued, so you can set up your trace patterns before.

`overload_check`

Allows to enable overload checking on the nodes under trace. `Module:Function(check)` is performed each `MSec` millisecond. If the check returns `true`, the tracing is disabled on a specified node.

`Module:Function` must be able to handle at least three atoms: `init`, `check`, and `stop`. `init` and `stop` allows you to initialize and clean up the check environment.

When a node gets overloaded, it is not possible to issue `ttb:p/2` or any command from the `ttb:tp/2,3,4` family, as it would lead to inconsistent tracing state (different trace specifications on different nodes).

`flush`

Periodically flushes all file trace port clients (see `dbg:flush_trace_port/1`). When enabled, the buffers are freed each `MSec` millisecond. This option is not allowed with `{file, {local, File}}` tracing.

`{resume, FetchTimeout}`

Enables the autoresume feature. When enabled, remote nodes try to reconnect to the controlling node if they are restarted. The feature requires application Runtime_Tools to be started (so it has to be present in the `.boot` scripts if the traced nodes run with embedded Erlang). If this is not possible, resume can be performed manually by starting `Runtime_Tools` remotely using `rpc:call/4`.

`ttb` tries to fetch all logs from a reconnecting node before reinitializing the trace. This must finish within `FetchTimeout` milliseconds or is aborted.

By default, autostart information is stored in a file named `ttb_autostart.bin` on each node. If this is not desired (for example, on diskless nodes), a custom module handling autostart information storage and retrieval can be provided by specifying environment variable `ttb_autostart_module` for the application Runtime_Tools. The module must respond to the following API:

`write_config(Data) -> ok`

Stores the provided data for further retrieval. It is important to realize that the data storage used must not be affected by the node crash.

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

Retrieves configuration stored with `write_config(Data)`.

`delete_config() -> ok`

Deletes configuration stored with `write_config(Data)`. Notice that after this call any subsequent calls to `read_config` must return `{error, Error}`.

`resume` implies the default `FetchTimeout`, which is 10 seconds

`p(Item,Flags) -> Return`

Types:

```
Return = {ok,[{Item,MatchDesc}]}
Items = Item | [Item]
Item = pid() | port() | RegName | {global,GlobalRegName} | all | processes
| ports | existing | existing_processes | existing_ports | new |
new_processes | new_ports
RegName = atom()
GlobalRegName = term()
Flags = Flag | [Flag]
```

Sets the specified trace flags on the specified processes or ports. Flag `timestamp` is always turned on.

See the Reference Manual for module `dbg` for the possible trace flags. Parameter `MatchDesc` is the same as returned from `dbg:p/2`.

Processes can be specified as registered names, globally registered names, or process identifiers. Ports can be specified as registered names or port identifiers. If a registered name is specified, the flags are set on processes/ports with this name on all active nodes.

Issuing this command starts the timer for this trace if option `timer` is specified with `tracer/2`.

```
tp(Module [, Function [, Arity]], MatchSpec)
tp({Module, Function , Arity}, MatchSpec)
tpl(Module [, Function [, Arity]], MatchSpec)
tpl({Module, Function , Arity}, MatchSpec)
ctp()
ctp(Module [, Function [, Arity]])
ctp({Module, Function, Arity})
ctpl()
ctpl(Module [, Function [, Arity]])
ctpl({Module, Function, Arity})
ctpg()
ctpg(Module [, Function [, Arity]])
ctpg({Module, Function, Arity})
tpe(Event,MatchSpec)
ctpe(Event)
```

These functions are to be used with trace flag `call`, `send`, and `'receive'` for setting and clearing trace patterns.

When trace flag `call` is set on a process, function calls are traced on that process if a trace pattern is set for the called function.

The `send` and `'receive'` flags enable tracing of all messages sent and received by the process/port. Trace patterns set with `tpe` may limit traced messages based on the message content, the sender, and/or the receiver.

Trace patterns specify how to trace a function or a message by using match specifications. Match specifications are described in the `ERTS User's Guide`.

These functions are equivalent to the corresponding functions in module dbg, but all calls are stored in the history. The history buffer makes it easy to create configuration files; the same trace environment can be set up many times, for example, to compare two test runs. It also reduces the amount of typing when using `ttb` from the Erlang shell.

`tp`

Sets trace patterns on global function calls.

`tpl`

Sets trace patterns on local and global function calls.

`tpe`

Sets trace patterns on messages.

`ctp`

Clears trace patterns on local and global function calls.

`ctpl`

Clears trace patterns on local function calls.

`ctpg`

Clears trace patterns on global function calls.

`ctpe`

Clears trace patterns on messages.

With `tp` and `tpl`, one of the match specification shortcuts can be used (for example, `ttb:tp(foo_module, caller)`).

The shortcuts are as follows:

- `return` - for `[{'_',[],[{return_trace}]}]` (report the return value from a traced function)
- `caller` - for `[{'_',[],[{message,{caller}}]}]` (report the calling function)
- `{codestr, Str}` - for `dbg:fun2ms/1` arguments passed as strings (example: `"fun(_) -> return_trace() end"`)

`list_history() -> History`

Types:

`History = [{N,Func,Args}]`

All calls to `ttb` is stored in the history. This function returns the current content of the history. Any entry can be reexecuted with `run_history/1` or stored in a configuration file with `write_config/2,3`.

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

Types:

`N = integer() | [integer()]`

Executes the specified entry or entries from the history list. To list history, use `list_history/0`.

`write_config(ConfigFile,Config)`

Equivalent to `write_config(ConfigFile,Config,[])`.

write_config(ConfigFile,Config,Opts) -> ok | {error,Reason}

Types:

ConfigFile = string()
Config = all | [integer()] | [{Mod,Func,Args}]
Mod = atom()
Func = atom()
Args = [term()]
Opts = Opt | [Opt]
Opt = append

Creates or extends a configuration file, which can be used for restoring a specific configuration later.

The contents of the configuration file can either be fetched from the history or specified directly as a list of `{Mod,Func,Args}`.

If the complete history is to be stored in the configuration file, `Config` must be `all`. If only a selected number of entries from the history are to be stored, `Config` must be a list of integers pointing out the entries to be stored.

If `Opts` is not specified or if it is `[]`, `ConfigFile` is deleted and a new file is created. If `Opts = [append]`, `ConfigFile` is not deleted. The new information is appended at the end of the file.

run_config(ConfigFile) -> ok | {error,Reason}

Types:

ConfigFile = string()

Executes all entries in the specified configuration file. Notice that the history of the last trace is always available in file `ttb_last_config`.

run_config(ConfigFile,NumList) -> ok | {error,Reason}

Types:

ConfigFile = string()
NumList = [integer()]

Executes selected entries from the specified configuration file. `NumList` is a list of integers pointing out the entries to be executed.

To list the contents of a configuration file, use `list_config/1`.

Notice that the history of the last trace is always available in file `ttb_last_config`.

list_config(ConfigFile) -> Config | {error,Reason}

Types:

ConfigFile = string()
Config = [{N,Func,Args}]

Lists all entries in the specified configuration file.

write_trace_info(Key,Info) -> ok

Types:

Key = term()
Info = Data | fun() -> Data
Data = term()

File `.ti` contains `{Key,ValueList}` tuples. This function adds `Data` to the `ValueList` associated with `Key`. All information written with this function is included in the call to the format handler.

`seq_trigger_ms() -> MatchSpec`

Equivalent to `seq_trigger_ms(all)`.

`seq_trigger_ms(Flags) -> MatchSpec`

Types:

```
MatchSpec = match_spec()
Flags = all | SeqTraceFlag | [SeqTraceFlag]
SeqTraceFlag = atom()
```

A match specification can turn on or off sequential tracing. This function returns a match specification, which turns on sequential tracing with the specified `Flags`.

This match specification can be specified as the last argument to `tp` or `tpl`. The activated `Item` then becomes a **trigger** for sequential tracing. This means that if the item is called on a process with trace flag `call` set, the process is "contaminated" with token `seq_trace`.

If `Flags = all`, all possible flags are set.

The possible values for `SeqTraceFlag` are available in `seq_trace`.

For a description of the `match_spec()` syntax, see section `Match Specifications in Erlang` in ERTS, which explains the general match specification "language".

> **Note:**
>
> The **system tracer** for sequential tracing is automatically initiated by `ttb` when a trace port is started with `ttb:tracer/0,1,2`.

An example of how to use function `seq_trigger_ms/0,1` follows:

```
(tiger@durin)5> ttb:tracer().
{ok,[tiger@durin]}
(tiger@durin)6> ttb:p(all,call).
{ok,{[all],[call]}}
(tiger@durin)7> ttb:tp(mod,func,ttb:seq_trigger_ms()).
{ok,[{matched,1},{saved,1}]}
(tiger@durin)8>
```

Whenever `mod:func(...)` is called after this, token `seq_trace` is set on the executing process.

`stop()`

Equivalent to `stop([])`.

`stop(Opts) -> stopped | {stopped, Dir}`

Types:

```
Opts = Opt | [Opt]
Opt = nofetch | {fetch_dir, Dir} | format | {format, FormatOpts} |
return_fetch_dir
Dir = string()
```

FormatOpts = see format/2

Stops tracing on all nodes. Logs and trace information files are sent to the trace control node and stored in a directory named `ttb_upload_FileName-Timestamp`, where `Filename` is the one provided with `{file, File}` during trace setup and `Timestamp` is of the form `yyyymmdd-hhmmss`. Even logs from nodes on the same machine as the trace control node are moved to this directory. The history list is saved to a file named `ttb_last_config` for further reference (as it is no longer accessible through history and configuration management functions, like `ttb:list_history/0`).

Options:

`nofetch`

Indicates that trace logs are not to be collected after tracing is stopped.

`{fetch, Dir}`

Allows specification of the directory to fetch the data to. If the directory already exists, an error is thrown.

`format`

Indicates the trace logs to be formatted after tracing is stopped. All logs in the fetch directory are merged.

`return_fetch_dir`

Indicates the return value to be `{stopped, Dir}` and not just `stopped`. This implies `fetch`.

get_et_handler()

Returns the `et` handler, which can be used with `format/2` or `tracer/2`.

Example: `ttb:format(Dir, [{handler, ttb:get_et_handler()}])`.

format(File)

Equivalent to `format(File,[])`.

format(File,Options) -> ok | {error, Reason}

Types:

File = string() | [string()]

This can be the name of a binary log, a list of such logs, or the name of a directory containing one or more binary logs.

Options = Opt | [Opt]

Opt = {out,Out} | {handler,FormatHandler} | disable_sort

Out = standard_io | string()

FormatHandler = {Function, InitialState}

Function = fun(Fd,Trace,TraceInfo,State) -> State

Fd = standard_io | FileDescriptor

File descriptor of the destination file `Out`.

Trace = tuple()

The trace message. For details, see the Reference Manual for module `erlang`.

TraceInfo = [{Key,ValueList}]

Includes the keys `flags`, `client`, and `node`. If `handler` is specified as option to the tracer function, this is also included. Also, all information written with function `write_trace_info/2` is included.

Reads the specified binary trace log(s). The logs are processed in the order of their time stamps as long as option `disable_sort` is not specified.

If `FormatHandler = {Function,InitialState}`, `Function` is called for each trace message.

If `FormatHandler = get_et_handler()`, `et_viewer` in application ET is used for presenting the trace log graphically. `ttb` provides a few different filters that can be selected from menu **Filters and scaling** in the `et_viewer`.

If `FormatHandler` is not specified, a default handler is used presenting each trace message as a text line.

The state returned from each call of `Function` is passed to the next call, even if the next call is to format a message from another log file.

If `Out` is specified, `FormatHandler` gets the file descriptor to `Out` as the first parameter.

`Out` is ignored if the `et` format handler is used.

Wrap logs can be formatted one by one or all at once. To format one of the wrap logs in a set, specify the exact file name. To format the whole set of wrap logs, specify the name with `*` instead of the wrap count. For examples, see the `User's Guide`.

etop

Erlang module

Start Erlang Top with the provided scripts `etop`. This starts a hidden Erlang node that connects to the node to be measured. The measured node is specified with option `-node`. If the measured node has a different cookie than the default cookie for the user who invokes the script, the cookie must be explicitly specified with option `-setcookie`.

Under Windows, batch file `etop.bat` can be used.

When executing the `etop` script, configuration parameters can be specified as command-line options, for example, `etop -node testnode@myhost -setcookie MyCookie`. The following configuration parameters exist for the tool:

`node`

The measured node.

Value: `atom()`

Mandatory

`setcookie`

Cookie to use for the `etop` node. Must be same as the cookie on the measured node.

Value: `atom()`

`lines`

Number of lines (processes) to display.

Value: `integer()`

Default: `10`

`interval`

Time interval (in seconds) between each update of the display.

Value: `integer()`

Default: `5`

`accumulate`

If `true`, the execution time and reductions are accumulated.

Value: `boolean()`

Default: `false`

`sort`

Identifies what information to sort by.

Value: `runtime | reductions | memory | msg_q`

Default: `runtime` (`reductions` if `tracing=off`)

`tracing`

`etop` uses the Erlang trace facility, and thus no other tracing is possible on the measured node while `etop` is running, unless this option is set to `off`. Also helpful if the `etop` tracing causes too high load on the measured node. With tracing off, runtime is not measured.

Value: `on | off`

Default: on

For details about Erlang Top, see the User's Guide.

Exports

`start() -> ok`

Starts `etop`. Notice that `etop` is preferably started with the `etop` script.

`start(Options) -> ok`

Types:

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

Starts `etop`. To view the possible options, use `help/0`.

`help() -> ok`

Displays the help of `etop` and its options.

`config(Key,Value) -> Result`

Types:

```
Result = ok | {error,Reason}
Key = lines | interval | accumulate | sort
Value = term()
```

Changes the configuration parameters of the tool during runtime. Allowed parameters are `lines`, `interval`, `accumulate`, and `sort`.

`dump(File) -> Result`

Types:

```
Result = ok | {error,Reason}
File = string()
```

Dumps the current display to a text file.

`stop() -> stop`

Terminates `etop`.

crashdump_viewer

Erlang module

The Crashdump Viewer is a WxWidgets based tool for browsing Erlang crashdumps.

For details about how to get started with the Crashdump Viewer, see the `User's Guide`.

Exports

`start() -> ok`

`start(File) -> ok`

Types:

`File = string()`

The filename of the crashdump.

Starts the Crashdump Viewer GUI and loads the specified crashdump.

If `File` is not specified, a file dialog is opened where the crashdump can be selected.

`stop() -> ok`

Terminates the Crashdump Viewer and closes all GUI windows.

cdv

Command

The `cdv` shell script is located in directory `priv` of the Observer application. The script is used for starting the Crashdump Viewer tool from the OS command line.

For Windows users, `cdv.bat` is found in the same location.

Exports

`cdv [file]`

Argument `file` is optional. If not specified, a file dialog is displayed, allowing you to select a crashdump from the file system.