Runtime_Tools

Runtime_Tools 1.12.5
August 15, 2024

August 15, 2024

1 Runtime Tools User's Guide

Runtime Tools

1.1 LTTng and Erlang/OTP

1.1.1 Introduction

The Linux Trace Toolkit: next generation is an open source system software package for correlated tracing of the Linux kernel, user applications and libraries.

For more information, please visit **http://lttng.org**

1.1.2 Building Erlang/OTP with LTTng support

Configure and build Erlang with LTTng support:

For LTTng to work properly with Erlang/OTP you need the following packages installed:

- LTTng-tools: a command line interface to control tracing sessions.
- LTTng-UST: user space tracing library.

On Ubuntu this can be installed via `aptitude`:

```
$ sudo aptitude install lttng-tools liblttng-ust-dev
```

See **Installing LTTng** for more information on how to install LTTng on your system.

After LTTng is properly installed on the system Erlang/OTP can be built with LTTng support.

```
$ ./configure --with-dynamic-trace=lttng
$ make
```

1.1.3 Dyntrace Tracepoints

All tracepoints are in the domain of `org_erlang_dyntrace`

All Erlang types are the string equivalent in LTTng.

process_spawn

- `pid : string` :: Process ID. Ex. `"<0.131.0>"`
- `parent : string` :: Process ID. Ex. `"<0.131.0>"`
- `entry : string` :: Code Location. Ex. `"lists:sort/1"`

Available through *`erlang:trace/3`* with trace flag `procs` and `{tracer,dyntrace,[]}` as tracer module.

Example:

```
process_spawn: { cpu_id = 3 }, { pid = "<0.131.0>", parent = "<0.130.0>", entry = "erlang:apply/2" }
```

process_link

- `to : string` :: Process ID or Port ID. Ex. `"<0.131.0>"`
- `from : string` :: Process ID or Port ID. Ex. `"<0.131.0>"`

- `type : string` :: `"link" | "unlink"`

Available through *erlang:trace/3* with trace flag `procs` and `{tracer,dyntrace,[]}` as tracer module.

Example:

```
process_link: { cpu_id = 3 }, { from = "<0.130.0>", to = "<0.131.0>", type = "link" }
```

process_exit

- `pid : string` :: Process ID. Ex. `"<0.131.0>"`
- `reason : string` :: Exit reason. Ex. `"normal"`

Available through *erlang:trace/3* with trace flag `procs` and `{tracer,dyntrace,[]}` as tracer module.

Example:

```
process_exit: { cpu_id = 3 }, { pid = "<0.130.0>", reason = "normal" }
```

process_register

- `pid : string` :: Process ID. Ex. `"<0.131.0>"`
- `name : string` :: Registered name. Ex. `"error_logger"`
- `type : string` :: `"register" | "unregister"`

Example:

```
process_register: { cpu_id = 0 }, { pid = "<0.128.0>", name = "dyntrace_lttng_SUITE" type = "register" }
```

process_scheduled

- `pid : string` :: Process ID. Ex. `"<0.131.0>"`
- `entry : string` :: Code Location. Ex. `"lists:sort/1"`
- `type : string` :: `"in" | "out" | "in_exiting" | "out_exiting" | "out_exited"`

Available through *erlang:trace/3* with trace flag `running` and `{tracer,dyntrace,[]}` as tracer module.

Example:

```
process_scheduled: { cpu_id = 0 }, { pid = "<0.136.0>", entry = "erlang:apply/2", type = "in" }
```

port_open

- `pid : string` :: Process ID. Ex. `"<0.131.0>"`
- `driver : string` :: Driver name. Ex. `"efile"`
- `port : string` :: Port ID. Ex. `"#Port<0.1031>"`

Available through *erlang:trace/3* with trace flag `ports` and `{tracer,dyntrace,[]}` as tracer module.

Example:

```
port_open: { cpu_id = 5 }, { pid = "<0.131.0>", driver = "'/bin/sh -s unix:cmd'", port = "#Port<0.1887>" }
```

port_exit

- `port : string` :: Port ID. Ex. `"#Port<0.1031>"`
- `reason : string` :: Exit reason. Ex. `"normal"`

Available through *erlang:trace/3* with trace flag `ports` and `{tracer,dyntrace,[]}` as tracer module.

Example:

```
port_exit: { cpu_id = 5 }, { port = "#Port<0.1887>", reason = "normal" }
```

port_link

- `to : string` :: Process ID. Ex. `"<0.131.0>"`
- `from : string` :: Process ID. Ex. `"<0.131.0>"`
- `type : string` :: `"link" | "unlink"`

Available through *erlang:trace/3* with trace flag `ports` and `{tracer,dyntrace,[]}` as tracer module.

Example:

```
port_link: { cpu_id = 5 }, { from = "#Port<0.1887>", to = "<0.131.0>", type = "unlink" }
```

port_scheduled

Available through *erlang:trace/3* with trace flag `running` and `{tracer,dyntrace,[]}` as tracer module.

- `port : string` :: Port ID. Ex. `"#Port<0.1031>"`
- `entry : string` :: Callback. Ex. `"open"`
- `type : string` :: `"in" | "out" | "in_exiting" | "out_exiting" | "out_exited"`

Example:

```
port_scheduled: { cpu_id = 5 }, { pid = "#Port<0.1905>", entry = "close", type = "out" }
```

Available through *erlang:trace/3* with trace flag `running` and `{tracer,dyntrace,[]}` as tracer module.

function_call

- `pid : string` :: Process ID. Ex. `"<0.131.0>"`
- `entry : string` :: Code Location. Ex. `"lists:sort/1"`
- `depth : integer` :: Stack depth. Ex. `0`

Available through *erlang:trace/3* with trace flag `call` and `{tracer,dyntrace,[]}` as tracer module.

Example:

```
function_call: { cpu_id = 5 }, { pid = "<0.145.0>", entry = "dyntrace_lttng_SUITE:'-t_call/1-fun-1-'/0", dep
```

function_return

- `pid : string` :: Process ID. Ex. `"<0.131.0>"`
- `entry : string` :: Code Location. Ex. `"lists:sort/1"`
- `depth : integer` :: Stack depth. Ex. `0`

Available through *erlang:trace/3* with trace flag `call` or `return_to` and `{tracer,dyntrace,[]}` as tracer module.

Example:

```
function_return: { cpu_id = 5 }, { pid = "<0.145.0>", entry = "dyntrace_lttng_SUITE:waiter/0", depth = 0 }
```

function_exception

- `pid : string` :: Process ID. Ex. `"<0.131.0>"`
- `entry : string` :: Code Location. Ex. `"lists:sort/1"`
- `class : string` :: Error reason. Ex. `"error"`

Available through *erlang:trace/3* with trace flag `call` and `{tracer,dyntrace,[]}` as tracer module.

Example:

```
function_exception: { cpu_id = 5 }, { pid = "<0.144.0>", entry = "t:call_exc/1", class = "error" }
```

message_send

- `from : string` :: Process ID or Port ID. Ex. `"<0.131.0>"`
- `to : string` :: Process ID or Port ID. Ex. `"<0.131.0>"`
- `message : string` :: Message sent. Ex. `"{<0.162.0>,ok}"`

Available through *erlang:trace/3* with trace flag `send` and `{tracer,dyntrace,[]}` as tracer module.

Example:

```
message_send: { cpu_id = 3 }, { from = "#Port<0.1938>", to = "<0.160.0>", message = "{#Port<0.1938>,eof}" }
```

message_receive

- `to : string` :: Process ID or Port ID. Ex. `"<0.131.0>"`
- `message : string` :: Message received. Ex. `"{<0.162.0>,ok}"`

Available through *erlang:trace/3* with trace flag `'receive'` and `{tracer,dyntrace,[]}` as tracer module.

Example:

```
message_receive: { cpu_id = 7 }, { to = "<0.167.0>", message = "{<0.165.0>,ok}" }
```

gc_minor_start

- `pid : string` :: Process ID. Ex. `"<0.131.0>"`
- `need : integer` :: Heap need. Ex. `2`
- `heap : integer` :: Young heap word size. Ex. `233`
- `old_heap : integer` :: Old heap word size. Ex. `233`

Available through *erlang:trace/3* with trace flag `garbage_collection` and `{tracer,dyntrace,[]}` as tracer module.

Example:

```
gc_minor_start: { cpu_id = 0 }, { pid = "<0.172.0>", need = 0, heap = 610, old_heap = 0 }
```

gc_minor_end

- `pid : string` :: Process ID. Ex. `"<0.131.0>"`
- `reclaimed : integer` :: Heap reclaimed. Ex. `2`
- `heap : integer` :: Young heap word size. Ex. `233`
- `old_heap : integer` :: Old heap word size. Ex. `233`

Available through *erlang:trace/3* with trace flag `garbage_collection` and `{tracer,dyntrace,[]}` as tracer module.

Example:

```
gc_minor_end: { cpu_id = 0 }, { pid = "<0.172.0>", reclaimed = 120, heap = 1598, old_heap = 1598 }
```

gc_major_start

- `pid : string` :: Process ID. Ex. `"<0.131.0>"`
- `need : integer` :: Heap need. Ex. `2`
- `heap : integer` :: Young heap word size. Ex. `233`
- `old_heap : integer` :: Old heap word size. Ex. `233`

Available through *erlang:trace/3* with trace flag `garbage_collection` and `{tracer,dyntrace,[]}` as tracer module.

Example:

```
gc_major_start: { cpu_id = 0 }, { pid = "<0.172.0>", need = 8, heap = 2586, old_heap = 1598 }
```

gc_major_end

- `pid : string` :: Process ID. Ex. `"<0.131.0>"`
- `reclaimed : integer` :: Heap reclaimed. Ex. `2`
- `heap : integer` :: Young heap word size. Ex. `233`
- `old_heap : integer` :: Old heap word size. Ex. `233`

Available through *erlang:trace/3* with trace flag `garbage_collection` and `{tracer,dyntrace,[]}` as tracer module.

Example:

```
gc_major_end: { cpu_id = 0 }, { pid = "<0.172.0>", reclaimed = 240, heap = 4185, old_heap = 0 }
```

1.1.4 BEAM Tracepoints

All tracepoints are in the domain of `org_erlang_otp`

All Erlang types are the string equivalent in LTTng.

scheduler_poll

- `scheduler : integer` :: Scheduler ID. Ex. `1`
- `runnable : integer` :: Runnable. Ex. `1`

Example:

```
scheduler_poll: { cpu_id = 4 }, { scheduler = 1, runnable = 1 }
```

driver_init

- `driver : string` :: Driver name. Ex. `"efile"`
- `major : integer` :: Major version. Ex. `3`
- `minor : integer` :: Minor version. Ex. `1`
- `flags : integer` :: Flags. Ex. `1`

Example:

```
driver_init: { cpu_id = 2 }, { driver = "caller_drv", major = 3, minor = 3, flags = 1 }
```

driver_start

- `pid : string` :: Process ID. Ex. `"<0.131.0>"`
- `driver : string` :: Driver name. Ex. `"efile"`
- `port : string` :: Port ID. Ex. `"#Port<0.1031>"`

Example:

```
driver_start: { cpu_id = 2 }, { pid = "<0.198.0>", driver = "caller_drv", port = "#Port<0.3676>" }
```

driver_output

- `pid : string` :: Process ID. Ex. `"<0.131.0>"`

- `port : string` :: Port ID. Ex. `"#Port<0.1031>"`
- `driver : string` :: Driver name. Ex. `"efile"`
- `bytes : integer` :: Size of data returned. Ex. `82`

Example:

```
driver_output: { cpu_id = 2 }, { pid = "<0.198.0>", port = "#Port<0.3677>", driver = "/bin/sh -s unix:cmd", byt
```

driver_outputv

- `pid : string` :: Process ID. Ex. `"<0.131.0>"`
- `port : string` :: Port ID. Ex. `"#Port<0.1031>"`
- `driver : string` :: Driver name. Ex. `"efile"`
- `bytes : integer` :: Size of data returned. Ex. `82`

Example:

```
driver_outputv: { cpu_id = 5 }, { pid = "<0.194.0>", port = "#Port<0.3663>", driver = "tcp_inet", bytes = 3 }
```

driver_ready_input

- `pid : string` :: Process ID. Ex. `"<0.131.0>"`
- `port : string` :: Port ID. Ex. `"#Port<0.1031>"`
- `driver : string` :: Driver name. Ex. `"efile"`

Example:

```
driver_ready_input: { cpu_id = 5 }, { pid = "<0.189.0>", port = "#Port<0.3637>", driver = "inet_gethost 4 " }
```

driver_ready_output

- `pid : string` :: Process ID. Ex. `"<0.131.0>"`
- `port : string` :: Port ID. Ex. `"#Port<0.1031>"`
- `driver : string` :: Driver name. Ex. `"efile"`

Example:

```
driver_ready_output: { cpu_id = 5 }, { pid = "<0.194.0>", port = "#Port<0.3663>", driver = "tcp_inet" }
```

driver_timeout

- `pid : string` :: Process ID. Ex. `"<0.131.0>"`
- `port : string` :: Port ID. Ex. `"#Port<0.1031>"`
- `driver : string` :: Driver name. Ex. `"efile"`

Example:

```
driver_timeout: { cpu_id = 5 }, { pid = "<0.196.0>", port = "#Port<0.3664>", driver = "tcp_inet" }
```

driver_stop_select

- `driver : string` :: Driver name. Ex. `"efile"`

Example:

```
driver_stop_select: { cpu_id = 5 }, { driver = "unknown" }
```

driver_flush

- `pid : string` :: Process ID. Ex. `"<0.131.0>"`

- port : string :: Port ID. Ex. "#Port<0.1031>"
- driver : string :: Driver name. Ex. "efile"

Example:

```
driver_flush: { cpu_id = 7 }, { pid = "<0.204.0>", port = "#Port<0.3686>", driver = "tcp_inet" }
```

driver_stop

- pid : string :: Process ID. Ex. "<0.131.0>"
- port : string :: Port ID. Ex. "#Port<0.1031>"
- driver : string :: Driver name. Ex. "efile"

Example:

```
driver_stop: { cpu_id = 5 }, { pid = "[]", port = "#Port<0.3673>", driver = "efile" }
```

driver_process_exit

- pid : string :: Process ID. Ex. "<0.131.0>"
- port : string :: Port ID. Ex. "#Port<0.1031>"
- driver : string :: Driver name. Ex. "efile"

driver_ready_async

- pid : string :: Process ID. Ex. "<0.131.0>"
- port : string :: Port ID. Ex. "#Port<0.1031>"
- driver : string :: Driver name. Ex. "efile"

Example:

```
driver_ready_async: { cpu_id = 3 }, { pid = "<0.181.0>", port = "#Port<0.3622>", driver = "efile" }
```

driver_call

- pid : string :: Process ID. Ex. "<0.131.0>"
- port : string :: Port ID. Ex. "#Port<0.1031>"
- driver : string :: Driver name. Ex. "efile"
- command : integer :: Command integer. Ex. 1
- bytes : integer :: Size of data returned. Ex. 82

Example:

```
driver_call: { cpu_id = 2 }, { pid = "<0.202.0>", port = "#Port<0.3676>", driver = "caller_drv", command =
```

driver_control

- pid : string :: Process ID. Ex. "<0.131.0>"
- port : string :: Port ID. Ex. "#Port<0.1031>"
- driver : string :: Driver name. Ex. "efile"
- command : integer :: Command integer. Ex. 1
- bytes : integer :: Size of data returned. Ex. 82

Example:

```
driver_control: { cpu_id = 3 }, { pid = "<0.32767.8191>", port = "#Port<0.0>", driver = "forker", command =
```

aio_pool_get

- port : string :: Port ID. Ex. "#Port<0.1031>"
- length : integer :: Async queue length. Ex. 0

Example:

```
aio_pool_get: { cpu_id = 4 }, { port = "#Port<0.3614>", length = 0 }
```

aio_pool_put

- port : string :: Port ID. Ex. "#Port<0.1031>"
- length : integer :: Async queue length. Ex. -1

Async queue length is not defined for put operations.

Example:

```
aio_pool_put: { cpu_id = 3 }, { port = "#Port<0.3614>", length = -1 }
```

carrier_create

- type : string :: Carrier type. Ex. "ets_alloc"
- instance : integer :: Allocator instance. Ex. 1
- size : integer :: Carrier size. Ex. 262144
- mbc_carriers : integer :: Number of multiblock carriers in instance. Ex. 3
- mbc_carriers_size : integer :: Total size of multiblock blocks carriers in instance. Ex. 1343488
- mbc_blocks : integer :: Number of multiblock blocks in instance. Ex. 122
- mbc_blocks_size : integer :: Total size of all multiblock blocks in instance. Ex. 285296
- sbc_carriers : integer :: Number of singleblock carriers in instance. Ex. 1
- sbc_carriers_size : integer :: Total size of singleblock blocks carriers in instance. Ex. 1343488
- sbc_blocks : integer :: Number of singleblocks in instance. Ex. 1
- sbc_blocks_size : integer :: Total size of all singleblock blocks in instance. Ex. 285296

Example:

```
carrier_create: { cpu_id = 2 }, { type = "ets_alloc", instance = 7, size = 2097152, mbc_carriers = 4, mbc_carri
```

carrier_destroy

- type : string :: Carrier type. Ex. "ets_alloc"
- instance : integer :: Allocator instance. Ex. 1
- size : integer :: Carrier size. Ex. 262144
- mbc_carriers : integer :: Number of multiblock carriers in instance. Ex. 3
- mbc_carriers_size : integer :: Total size of multiblock blocks carriers in instance. Ex. 1343488
- mbc_blocks : integer :: Number of multiblock blocks in instance. Ex. 122
- mbc_blocks_size : integer :: Total size of all multiblock blocks in instance. Ex. 285296
- sbc_carriers : integer :: Number of singleblock carriers in instance. Ex. 1
- sbc_carriers_size : integer :: Total size of singleblock blocks carriers in instance. Ex. 1343488
- sbc_blocks : integer :: Number of singleblocks in instance. Ex. 1
- sbc_blocks_size : integer :: Total size of all singleblock blocks in instance. Ex. 285296

Example:

```
carrier_destroy: { cpu_id = 6 }, { type = "ets_alloc", instance = 7, size = 262144, mbc_carriers = 3, mbc_carri
```

carrier_pool_put

- `type : string` :: Carrier type. Ex. `"ets_alloc"`
- `instance : integer` :: Allocator instance. Ex. `1`
- `size : integer` :: Carrier size. Ex. `262144`

Example:

```
carrier_pool_put: { cpu_id = 3 }, { type = "ets_alloc", instance = 5, size = 1048576 }
```

carrier_pool_get

- `type : string` :: Carrier type. Ex. `"ets_alloc"`
- `instance : integer` :: Allocator instance. Ex. `1`
- `size : integer` :: Carrier size. Ex. `262144`

Example:

```
carrier_pool_get: { cpu_id = 7 }, { type = "ets_alloc", instance = 4, size = 3208 }
```

1.1.5 Example of process tracing

An example of process tracing of `os_mon` and friends.

Clean start of lttng in a bash shell.

```
$ lttng create erlang-demo
Spawning a session daemon
Session erlang-demo created.
Traces will be written in /home/egil/lttng-traces/erlang-demo-20160526-165920
```

Start an Erlang node with lttng enabled.

```
$ erl
Erlang/OTP 19 [erts-8.0] [source-4d7b24d] [64-bit] [smp:8:8] [async-threads:10] [hipe] [kernel-poll:false] [

Eshell V8.0  (abort with ^G)
1>
```

Load the `dyntrace` module.

```
1> l(dyntrace).
{module,dyntrace}
```

All tracepoints via dyntrace are now visible and can be listed through `lttng list -u`.

Enable the process_register LTTng tracepoint for Erlang.

```
$ lttng enable-event -u org_erlang_dyntrace:process_register
UST event org_erlang_dyntrace:process_register created in channel channel0
```

Enable process tracing for new processes and use `dyntrace` as tracer backend.

```
2> erlang:trace(new,true,[procs,{tracer,dyntrace,[]}]).
0
```

Start LTTng tracing.

```
$ lttng start
Tracing started for session erlang-demo
```

Start the `os_mon` application in Erlang.

```
3> application:ensure_all_started(os_mon).
{ok,[sasl,os_mon]}
```

Stop LTTng tracing and view the result.

```
$ lttng stop
Tracing stopped for session erlang-demo
$ lttng view
[17:20:42.561168759] (+?.?????????) elxd1168lx9 org_erlang_dyntrace:process_register: \
    { cpu_id = 5 }, { pid = "<0.66.0>", name = "sasl_sup", type = "register" }
[17:20:42.561215519] (+0.000046760) elxd1168lx9 org_erlang_dyntrace:process_register: \
    { cpu_id = 5 }, { pid = "<0.67.0>", name = "sasl_safe_sup", type = "register" }
[17:20:42.562149024] (+0.000933505) elxd1168lx9 org_erlang_dyntrace:process_register: \
    { cpu_id = 5 }, { pid = "<0.68.0>", name = "alarm_handler", type = "register" }
[17:20:42.571035803] (+0.008886779) elxd1168lx9 org_erlang_dyntrace:process_register: \
    { cpu_id = 5 }, { pid = "<0.69.0>", name = "release_handler", type = "register" }
[17:20:42.574939868] (+0.003904065) elxd1168lx9 org_erlang_dyntrace:process_register: \
    { cpu_id = 5 }, { pid = "<0.74.0>", name = "os_mon_sup", type = "register" }
[17:20:42.576818712] (+0.001878844) elxd1168lx9 org_erlang_dyntrace:process_register: \
    { cpu_id = 5 }, { pid = "<0.75.0>", name = "disksup", type = "register" }
[17:20:42.580032013] (+0.003213301) elxd1168lx9 org_erlang_dyntrace:process_register: \
    { cpu_id = 5 }, { pid = "<0.76.0>", name = "memsup", type = "register" }
[17:20:42.583046339] (+0.003014326) elxd1168lx9 org_erlang_dyntrace:process_register: \
    { cpu_id = 5 }, { pid = "<0.78.0>", name = "cpu_sup", type = "register" }
[17:20:42.586206242] (+0.003159903) elxd1168lx9 org_erlang_dyntrace:process_register: \
    { cpu_id = 5 }, { pid = "<0.82.0>", name = "timer_server", type = "register" }
```

1.2 DTrace and Erlang/OTP

1.2.1 History

The first implementation of DTrace probes for the Erlang virtual machine was presented at the **2008 Erlang User Conference**. That work, based on the Erlang/OTP R12 release, was discontinued due to what appears to be miscommunication with the original developers.

Several users have created Erlang port drivers, linked-in drivers, or NIFs that allow Erlang code to try to activate a probe, e.g. `foo_module:dtrace_probe("message goes here!")`.

1.2.2 Goals

- Annotate as much of the Erlang VM as is practical.
- The initial goal is to trace file I/O operations.
- Support all platforms that implement DTrace: OS X, Solaris, and (I hope) FreeBSD and NetBSD.
- To the extent that it's practical, support SystemTap on Linux via DTrace provider compatibility.
- Allow Erlang code to supply annotations.

1.2.3 Supported platforms

- OS X 10.6.x / Snow Leopard, OS X 10.7.x / Lion and probably newer versions.
- Solaris 10. I have done limited testing on Solaris 11 and OpenIndiana release 151a, and both appear to work.
- FreeBSD 9.0 and 10.0.
- Linux via SystemTap compatibility. Please see *$ERL_TOP/HOWTO/SYSTEMTAP.md* for more details.

Just add the `--with-dynamic-trace=dtrace` option to your command when you run the `configure` script. If you are using systemtap, the configure option is `--with-dynamic-trace=systemtap`

1.2.4 Status

As of R15B01, the dynamic trace code is included in the OTP source distribution, although it's considered experimental. The main development of the dtrace code still happens outside of Ericsson, but there is no need to fetch a patched version of the OTP source to get the basic functionality.

1.2.5 Implementation summary

So far, most effort has been focused on the `efile_drv.c` code, which implements most file I/O on behalf of the Erlang virtual machine. This driver also presents a big challenge: its use of an I/O worker pool (enabled by using the `erl +A 8` flag, for example) makes it much more difficult to trace I/O activity because each of the following may be executed in a different Pthread:

- I/O initiation (Erlang code)
- I/O proxy process handling, e.g. read/write when file is not opened in `raw` mode, operations executed by the code & file server processes. (Erlang code)
- `efile_drv` command setup (C code)
- `efile_drv` command execution (C code)
- `efile_drv` status return (C code)

Example output from `lib/runtime_tools/examples/efile_drv.d` while executing `file:rename("old-name", "new-name")`:

```
efile_drv enter tag={3,84} user tag some-user-tag | RENAME (12) | args: old-name new-name ,\
      0 0 (port #Port<0.59>)
async I/O worker tag={3,83} | RENAME (12) | efile_drv-int_entry
async I/O worker tag={3,83} | RENAME (12) | efile_drv-int_return
efile_drv return tag={3,83} user tag  | RENAME (12) | errno 2
```

... where the following key can help decipher the output:

- `{3,83}` is the Erlang scheduler thread number (3) and operation counter number (83) assigned to this I/O operation. Together, these two numbers form a unique ID for the I/O operation.
- `12` is the command number for the rename operation. See the definition for `FILE_RENAME` in the source code file `efile_drv.c` or the `BEGIN` section of the D script `lib/runtime_tools/examples/efile_drv.d`.
- `old-name` and `new-name` are the two string arguments for the source and destination of the `rename(2)` system call. The two integer arguments are unused; the simple formatting code prints the arguments anyway, 0 and 0.
- The worker pool code was called on behalf of Erlang port `#Port<0.59>`.
- The system call failed with a POSIX errno value of 2: `ENOENT`, because the path `old-name` does not exist.
- The `efile_drv-int_entry` and `efile_drv_int_return` probes are provided in case the user is interested in measuring only the latency of code executed by `efile_drv` asynchronous functions by I/O worker pool threads and the OS system call that they encapsulate.

So, where does the `some-user-tag` string come from?

At the moment, the user tag comes from code like the following:

```
dyntrace:put_tag("some-user-tag"),
file:rename("old-name", "new-name"),
```

This method of tagging I/O at the Erlang level is subject to change.

1.2.6 Example DTrace probe specification

```
/**
 * Fired when a message is sent from one local process to another.
 *
 * NOTE: The 'size' parameter is in machine-dependent words and
 *       that the actual size of any binary terms in the message
 *       are not included.
 *
 * @param sender the PID (string form) of the sender
 * @param receiver the PID (string form) of the receiver
 * @param size the size of the message being delivered (words)
 * @param token_label for the sender's sequential trace token
 * @param token_previous count for the sender's sequential trace token
 * @param token_current count for the sender's sequential trace token
 */
probe message__send(char *sender, char *receiver, uint32_t size,
                    int token_label, int token_previous, int token_current);

/**
 * Fired when a message is sent from a local process to a remote process.
 *
 * NOTE: The 'size' parameter is in machine-dependent words and
 *       that the actual size of any binary terms in the message
 *       are not included.
 *
 * @param sender the PID (string form) of the sender
 * @param node_name the Erlang node name (string form) of the receiver
 * @param receiver the PID/name (string form) of the receiver
 * @param size the size of the message being delivered (words)
 * @param token_label for the sender's sequential trace token
 * @param token_previous count for the sender's sequential trace token
 * @param token_current count for the sender's sequential trace token
 */
probe message__send__remote(char *sender, char *node_name, char *receiver,
                            uint32_t size,
                    int token_label, int token_previous, int token_current);

/**
 * Fired when a message is queued to a local process.  This probe
 * will not fire if the sender's pid == receiver's pid.
 *
 * NOTE: The 'size' parameter is in machine-dependent words and
 *       that the actual size of any binary terms in the message
 *       are not included.
 *
 * @param receiver the PID (string form) of the receiver
 * @param size the size of the message being delivered (words)
 * @param queue_len length of the queue of the receiving process
 * @param token_label for the sender's sequential trace token
 * @param token_previous count for the sender's sequential trace token
 * @param token_current count for the sender's sequential trace token
 */
probe message__queued(char *receiver, uint32_t size, uint32_t queue_len,
                    int token_label, int token_previous, int token_current);

/**
 * Fired when a message is 'receive'd by a local process and removed
 * from its mailbox.
 *
 * NOTE: The 'size' parameter is in machine-dependent words and
 *       that the actual size of any binary terms in the message
 *       are not included.
 *
```

```
 * @param receiver the PID (string form) of the receiver
 * @param size the size of the message being delivered (words)
 * @param queue_len length of the queue of the receiving process
 * @param token_label for the sender's sequential trace token
 * @param token_previous count for the sender's sequential trace token
 * @param token_current count for the sender's sequential trace token
 */
probe message__receive(char *receiver, uint32_t size, uint32_t queue_len,
                    int token_label, int token_previous, int token_current);

/* ... */

/* Async driver pool */

/**
 * Show the post-add length of the async driver thread pool member's queue.
 *
 * NOTE: The port name is not available: additional lock(s) must
 *       be acquired in order to get the port name safely in an SMP
 *       environment.  The same is true for the aio__pool_get probe.
 *
 * @param port the Port (string form)
 * @param new queue length
 */
probe aio_pool__add(char *, int);

/**
 * Show the post-get length of the async driver thread pool member's queue.
 *
 * @param port the Port (string form)
 * @param new queue length
 */
probe aio_pool__get(char *, int);

/* Probes for efile_drv.c */

/**
 * Entry into the efile_drv.c file I/O driver
 *
 * For a list of command numbers used by this driver, see the section
 * "Guide to probe arguments" in ../../../README.md.  That section
 * also contains explanation of the various integer and string
 * arguments that may be present when any particular probe fires.
 *
 * TODO: Adding the port string, args[10], is a pain.  Making that
 *       port string available to all the other efile_drv.c probes
 *       will be more pain.  Is the pain worth it?  If yes, then
 *       add them everywhere else and grit our teeth.  If no, then
 *       rip it out.
 *
 * @param thread-id number of the scheduler Pthread                   arg0
 * @param tag number: {thread-id, tag} uniquely names a driver operation
 * @param user-tag string                                              arg2
 * @param command number                                               arg3
 * @param string argument 1                                            arg4
 * @param string argument 2                                            arg5
 * @param integer argument 1                                           arg6
 * @param integer argument 2                                           arg7
 * @param integer argument 3                                           arg8
 * @param integer argument 4                                           arg9
 * @param port the port ID of the busy port                        args[10]
 */
probe efile_drv__entry(int, int, char *, int, char *, char *,
                       int64_t, int64_t, int64_t, int64_t, char *);
```

```
/**
 * Entry into the driver's internal work function.  Computation here
 * is performed by a async worker pool Pthread.
 *
 * @param thread-id number
 * @param tag number
 * @param command number
 */
probe efile_drv__int_entry(int, int, int);

/**
 * Return from the driver's internal work function.
 *
 * @param thread-id number
 * @param tag number
 * @param command number
 */
probe efile_drv__int_return(int, int, int);

/**
 * Return from the efile_drv.c file I/O driver
 *
 * @param thread-id number                                          arg0
 * @param tag number                                                arg1
 * @param user-tag string                                           arg2
 * @param command number                                            arg3
 * @param Success? 1 is success, 0 is failure                       arg4
 * @param If failure, the errno of the error.                       arg5
 */
probe efile_drv__return(int, int, char *, int, int, int);
```

1.2.7 Guide to efile_drv.c probe arguments

```
/* Driver op code: used by efile_drv-entry      arg3 */
/*                 used by efile_drv-int_entry  arg3 */
/*                 used by efile_drv-int_return arg3 */
/*                 used by efile_drv-return     arg3 */

#define FILE_OPEN            1                 (probe arg3)
        probe arg6 = C driver dt_i1 = flags;
        probe arg4 = C driver dt_s1 = path;

#define FILE_READ            2                 (probe arg3)
        probe arg6 = C driver dt_i1 = fd;
        probe arg7 = C driver dt_i2 = flags;
        probe arg8 = C driver dt_i3 = size;

#define FILE_LSEEK           3                 (probe arg3)
        probe arg6 = C driver dt_i1 = fd;
        probe arg7 = C driver dt_i2 = offset;
        probe arg8 = C driver dt_i3 = origin;

#define FILE_WRITE           4                 (probe arg3)
        probe arg6 = C driver dt_i1 = fd;
        probe arg7 = C driver dt_i2 = flags;
        probe arg8 = C driver dt_i3 = size;

#define FILE_FSTAT           5                 (probe arg3)
        probe arg6 = C driver dt_i1 = fd;

#define FILE_PWD             6                 (probe arg3)
        none

#define FILE_READDIR         7                 (probe arg3)
        probe arg4 = C driver dt_s1 = path;

#define FILE_CHDIR           8                 (probe arg3)
        probe arg4 = C driver dt_s1 = path;

#define FILE_FSYNC           9                 (probe arg3)
            probe arg6 = C driver dt_i1 = fd;

#define FILE_MKDIR          10                 (probe arg3)
        probe arg4 = C driver dt_s1 = path;

#define FILE_DELETE         11                 (probe arg3)
        probe arg4 = C driver dt_s1 = path;

#define FILE_RENAME         12                 (probe arg3)
        probe arg4 = C driver dt_s1 = old_name;
        probe arg5 = C driver dt_s2 = new_name;

#define FILE_RMDIR          13                 (probe arg3)
        probe arg4 = C driver dt_s1 = path;

#define FILE_TRUNCATE       14                 (probe arg3)
        probe arg6 = C driver dt_i1 = fd;
        probe arg7 = C driver dt_i2 = flags;

#define FILE_READ_FILE      15                 (probe arg3)
        probe arg4 = C driver dt_s1 = path;

#define FILE_WRITE_INFO     16                 (probe arg3)
        probe arg6 = C driver dt_i1 = mode;
        probe arg7 = C driver dt_i2 = uid;
```

```
        probe arg8 = C driver dt_i3 = gid;

#define FILE_LSTAT          19                  (probe arg3)
        probe arg4 = C driver dt_s1 = path;

#define FILE_READLINK       20                  (probe arg3)
        probe arg4 = C driver dt_s1 = path;

#define FILE_LINK           21                  (probe arg3)
        probe arg4 = C driver dt_s1 = existing_path;
        probe arg5 = C driver dt_s2 = new_path;

#define FILE_SYMLINK        22                  (probe arg3)
        probe arg4 = C driver dt_s1 = existing_path;
        probe arg5 = C driver dt_s2 = new_path;

#define FILE_CLOSE          23                  (probe arg3)
        probe arg6 = C driver dt_i1 = fd;
        probe arg7 = C driver dt_i2 = flags;

#define FILE_PWRITEV        24                  (probe arg3)
        probe arg6 = C driver dt_i1 = fd;
        probe arg7 = C driver dt_i2 = flags;
        probe arg8 = C driver dt_i3 = size;

#define FILE_PREADV         25                  (probe arg3)
        probe arg6 = C driver dt_i1 = fd;
        probe arg7 = C driver dt_i2 = flags;
        probe arg8 = C driver dt_i3 = size;

#define FILE_SETOPT         26                  (probe arg3)
        probe arg6 = C driver dt_i1 = opt_name;
        probe arg7 = C driver dt_i2 = opt_specific_value;

#define FILE_IPREAD         27                  (probe arg3)
        probe arg6 = C driver dt_i1 = fd;
        probe arg7 = C driver dt_i2 = flags;
        probe arg8 = C driver dt_i3 = offsets[0];
        probe arg9 = C driver dt_i4 = size;

#define FILE_ALTNAME        28                  (probe arg3)
        probe arg4 = C driver dt_s1 = path;

#define FILE_READ_LINE      29                  (probe arg3)
        probe arg6 = C driver dt_i1 = fd;
        probe arg7 = C driver dt_i2 = flags;
        probe arg8 = C driver dt_i3 = read_offset;
        probe arg9 = C driver dt_i4 = read_ahead;

#define FILE_FDATASYNC      30                  (probe arg3)
        probe arg6 = C driver dt_i1 = fd;

#define FILE_FADVISE        31                  (probe arg3)
        probe arg6 = C driver dt_i1 = fd;
        probe arg7 = C driver dt_i2 = offset;
        probe arg8 = C driver dt_i3 = length;
        probe arg9 = C driver dt_i4 = advise_type;
```

1.3 SystemTap and Erlang/OTP

1.3.1 Introduction

SystemTap is DTrace for Linux. In fact Erlang's SystemTap support is built using SystemTap's DTrace compatibility's layer. For an introduction to Erlang DTrace support read *$ERL_TOP/HOWTO/DTRACE.md*.

1.3.2 Requisites

- Linux Kernel with UTRACE support

  check for UTRACE support in your current kernel:

  ```
  # grep CONFIG_UTRACE /boot/config-`uname -r`
  CONFIG_UTRACE=y
  ```

  Fedora 16 is known to contain UTRACE, for most other Linux distributions a custom build kernel will be required. Check Fedora's SystemTap documentation for additional required packages (e.g. Kernel Debug Symbols)
- SystemTap > 1.6

  A the time of writing this, the latest released version of SystemTap is version 1.6. Erlang's DTrace support requires a MACRO that was introduced after that release. So either get a newer release or build SystemTap from git yourself (see: http://sourceware.org/systemtap/getinvolved.html)

1.3.3 Building Erlang

Configure and build Erlang with SystemTap support:

```
# ./configure --with-dynamic-trace=systemtap + whatever args you need
# make
```

1.3.4 Testing

SystemTap, unlike DTrace, needs to know what binary it is tracing and has to be able to read that binary before it starts tracing. Your probe script therefore has to reference the correct beam emulator and stap needs to be able to find that binary. The examples are written for "beam", but other versions such as "beam.smp" or "beam.debug.smp" might exist (depending on your configuration). Make sure you either specify the full the path of the binary in the probe or your "beam" binary is in the search path.

All available probes can be listed like this:

```
# stap -L 'process("beam").mark("*")'
```

or:

```
# PATH=/path/to/beam:$PATH stap -L 'process("beam").mark("*")'
```

Probes in the dtrace.so NIF library like this:

```
# PATH=/path/to/dtrace/priv/lib:$PATH stap -L 'process("dtrace.so").mark("*")'
```

1.3.5 Running SystemTap scripts

Adjust the process("beam") reference to your beam version and attach the script to a running "beam" instance:

```
# stap /path/to/probe/script/port1.systemtap -x <pid of beam>
```

2 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, ports and messages.

To quickly get started on tracing function calls you can use the following code in the Erlang shell:

```
1> dbg:tracer(). %% Start the default trace message receiver
{ok,<0.36.0>}
2> dbg:p(all, c). %% Setup call (c) tracing on all processes
{ok,[{matched,nonode@nohost,26}]}
3> dbg:tp(lists, seq, x). %% Setup an exception return trace (x) on lists:seq
{ok,[{matched,nonode@nohost,2},{saved,x}]}
4> lists:seq(1,10).
(<0.34.0>) call lists:seq(1,10)
(<0.34.0>) returned from lists:seq/2 -> [1,2,3,4,5,6,7,8,9,10]
[1,2,3,4,5,6,7,8,9,10]
```

For more 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 match_spec
{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:

`pid()` or `port()`
: The corresponding process or port is traced. The process or port may be a remote process or port (on another Erlang node). The node must be in the list of traced nodes (see *n/1* and *tracer/3*).

`all`
: All processes and ports in the system as well as all processes and ports created hereafter are to be traced.

`processes`
: All processes in the system as well as all processes created hereafter are to be traced.

`ports`
: All ports in the system as well as all ports created hereafter are to be traced.

`new`
: All processes and ports created after the call is are to be traced.

`new_processes`
: All processes created after the call is are to be traced.

`new_ports`
: All ports created after the call is are to be traced.

`existing`
: All existing processes and ports are traced.

`existing_processes`
: All existing processes are traced.

`existing_ports`
: All existing ports are traced.

`atom()`
: The process or port with the corresponding registered name is traced. The process or port may be a remote process (on another Erlang node). The node must be added with the *n/1* or *tracer/3* function.

`integer()`
: The process <0.Item.0> is traced.

`{X, Y, Z}`
: The process <X.Y.Z> is traced.

`string()`
: If the `Item` is a string "<X.Y.Z>" as returned from *pid_to_list/1*, the process <X.Y.Z> is traced.

When enabling an `Item` that represents a group of processes, the `Item` is enabled on all nodes added with the *n/1* or *tracer/3* function.

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

`s (send)`

Traces the messages the process or port sends.

`r (receive)`

Traces the messages the process or port receives.

`m (messages)`

Traces the messages the process or port 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.

`ports`

Traces port related events to the port.

`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 except `silent`.

`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 and ports 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 and ports.

`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() | Built-inAlias | [] | match_spec()
Built-inAlias = x | c | cx
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 Specifications in Erlang*** explains the general match specification "language". The most common generic match specifications used can be found as `Built-inAlias`', see *ltp/0* below for details.

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/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 and ports 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 a couple of built-in aliases for common expressions, see *ltp/0* below for details.

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).

`tpe(Event, MatchSpec) -> {ok, MatchDesc} | {error, term()}`

Types:

`Event = send | 'receive'`
`MatchSpec = integer() | Built-inAlias | [] | match_spec()`
`Built-inAlias = x | c | cx`
`MatchDesc = [MatchInfo]`
`MatchInfo = {saved, integer()} | MatchNum`
`MatchNum = {matched, node(), 1} | {matched, node(), 0, RPCError}`

This function associates a match specification with trace event `send` or `'receive'`. By default all executed `send` and `'receive'` events are traced if enabled for a process. A match specification can be used to filter traced events based on sender, receiver and/or message content.

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 Specifications in Erlang*** explains the general match specification "language".

For `send`, the matching is done on the list `[Receiver, Msg]`. `Receiver` is the process or port identity of the receiver and `Msg` is the message term. The pid of the sending process can be accessed with the guard function `self/0`.

For `'receive'`, the matching is done on the list `[Node, Sender, Msg]`. `Node` is the node name of the sender. `Sender` is the process or port identity of the sender, or the atom `undefined` if the sender is not known (which may be the case for remote senders). `Msg` is the message term. The pid of the receiving process can be accessed with the guard function `self/0`.

All nodes added with *`n/1`* or *`tracer/3`* will be affected by this call.

The return value is the same as for *`tp/2`*. The number of matched events are never larger than 1 as `tpe/2` does not accept any form of wildcards for argument `Event`.

`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*).

ctpe(Event) -> {ok, MatchDesc} | {error, term()}

Types:

Event = send | 'receive'
MatchDesc = [MatchNum]
MatchNum = {matched, node(), 1} | {matched, node(), 0, RPCError}

This function clears match specifications for the specified trace event (`send` or `'receive'`). It will revert back to the default behavior of tracing all triggered events.

The return value follow the same style as for *ctp/1*.

`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*

There are three built-in trace patterns: `exception_trace`, `caller_trace` and `caller_exception_trace` (or `x`, `c` and `cx` respectively). Exception trace sets a trace which will show function names, parameters, return values and exceptions thrown from functions. Caller traces display function names, parameters and information about which function called it. An example using a built-in alias:

```
(x@y)4> dbg:tp(lists,sort,cx).
{ok,[{matched,nonode@nohost,2},{saved,cx}]}
(x@y)4> lists:sort([2,1]).
(<0.32.0>) call lists:sort([2,1]) ({erl_eval,do_apply,5})
(<0.32.0>) returned from lists:sort/1 -> [1,2]
[1,2]
```

`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() or port() as first argument, in which case the command is executed only on the node where the designated process or port 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 | module
Data = PortGenerator | HandlerSpec | ModuleSpec
PortGenerator = fun() (no arguments)
Error = term()
HandlerSpec = {HandlerFun, InitialData}
HandlerFun = fun() (two arguments)
ModuleSpec = fun() (no arguments) | {TracerModule, TracerState}
TracerModule = atom()
InitialData = TracerState = 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`), by a tracer port (`port`) or by a tracer module (`module`). For a description about tracer ports see *trace_port/2* and for a tracer modules see *erl_tracer*.

If `Type` is `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 choose any appropriate action to take when invoked, and can save a state for the next invocation by returning it.

If `Type` is `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 `Type` is `module`, then the second parameter should be either a tuple describing the *erl_tracer* module to be used for tracing and the state to be used for that tracer module or a fun returning the same tuple.

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. The default value of `QueSize` is 200.

`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 are allowed as well as their return values depend on which trace driver 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() | {module(), term()}`

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

`stop() -> ok`

Stops the `dbg` server, clears all trace flags for all processes, clears all trace patterns for all functions, clears trace patterns for send/receive, shuts down all trace clients, and closes all trace ports.

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)`.

dyntrace

Erlang module

This module implements interfaces to dynamic tracing, should such be compiled into the virtual machine. For a standard and/or commercial build, no dynamic tracing is available, in which case none of the functions in this module is usable or give any effect.

Should dynamic tracing be enabled in the current build, either by configuring with `./configure --with-dynamic-trace=dtrace` or with `./configure --with-dynamic-trace=systemtap`, the module can be used for two things:

- Trigger the user-probe `user_trace_i4s4` in the NIF library `dyntrace.so` by calling `dyntrace:p/{1,2,3,4,5,6,7,8}`.
- Set a user specified tag that will be present in the trace messages of both the `efile_drv` and the user-probe mentioned above.

Both building with dynamic trace probes and using them is experimental and unsupported by Erlang/OTP. It is included as an option for the developer to trace and debug performance issues in their systems.

The original implementation is mostly done by Scott Lystiger Fritchie as an Open Source Contribution and it should be viewed as such even though the source for dynamic tracing as well as this module is included in the main distribution. However, the ability to use dynamic tracing of the virtual machine is a very valuable contribution which OTP has every intention to maintain as a tool for the developer.

How to write `d` programs or `systemtap` scripts can be learned from books and from a lot of pages on the Internet. This manual page does not include any documentation about using the dynamic trace tools of respective platform. The `examples` directory of the `runtime_tools` application however contains comprehensive examples of both `d` and `systemtap` programs that will help you get started. Another source of information is the *dtrace* and *systemtap* chapters in the Runtime Tools Users' Guide.

Exports

`available() -> boolean()`

This function uses the NIF library to determine if dynamic tracing is available. Usually calling *erlang:system_info/1* is a better indicator of the availability of dynamic tracing.

The function will throw an exception if the `dyntrace` NIF library could not be loaded by the on_load function of this module.

`p() -> true | false | error | badarg`

Calling this function will trigger the "user" trace probe user_trace_i4s4 in the dyntrace NIF module, sending a trace message only containing the user tag and zeroes/empty strings in all other fields.

`p(integer() | string()) -> true | false | error | badarg`

Calling this function will trigger the "user" trace probe user_trace_i4s4 in the dyntrace NIF module, sending a trace message containing the user tag and the integer or string parameter in the first integer/string field.

`p(integer() | string(), integer() | string()) -> true | false | error | badarg`

Calling this function will trigger the "user" trace probe user_trace_i4s4 in the dyntrace NIF module, sending a trace message containing the user tag and the integer() or string() parameters as the first fields of respective type. integer() parameters should be put before any string() parameters. I.e. `p(1,"Hello")` is ok, as is `p(1,1)` and `p("Hello","Again")`, but not `p("Hello",1)`.

`p(integer() | string(), integer() | string(), integer() | string()) -> true | false | error | badarg`

Calling this function will trigger the "user" trace probe user_trace_i4s4 in the dyntrace NIF module, sending a trace message containing the user tag and the integer() or string() parameters as the first fields of respective type. integer() parameters should be put before any string() parameters, as in *p/2*.

`p(integer() | string(), integer() | string(), integer() | string(), integer() | string()) -> true | false | error | badarg`

Calling this function will trigger the "user" trace probe user_trace_i4s4 in the dyntrace NIF module, sending a trace message containing the user tag and the integer() or string() parameters as the first fields of respective type. integer() parameters should be put before any string() parameters, as in *p/2*.

`p(integer(), integer() | string(), integer() | string(), integer() | string(), string()) -> true | false | error | badarg`

Calling this function will trigger the "user" trace probe user_trace_i4s4 in the dyntrace NIF module, sending a trace message containing the user tag and the integer() or string() parameters as the first fields of respective type. integer() parameters should be put before any string() parameters, as in *p/2*.

There can be no more than four parameters of any type (integer() or string()), so the first parameter has to be an integer() and the last a string().

`p(integer(), integer(), integer() | string(), integer() | string(), string(), string()) -> true | false | error | badarg`

Calling this function will trigger the "user" trace probe user_trace_i4s4 in the dyntrace NIF module, sending a trace message containing the user tag and the integer() or string() parameters as the first fields of respective type. integer() parameters should be put before any string() parameters, as in *p/2*.

There can be no more than four parameters of any type (integer() or string()), so the first two parameters has to be integer()'s and the last two string()'s.

`p(integer(), integer(), integer(), integer() | string(), string(), string(), string()) -> true | false | error | badarg`

Calling this function will trigger the "user" trace probe user_trace_i4s4 in the dyntrace NIF module, sending a trace message containing the user tag and the integer() or string() parameters as the first fields of respective type. integer() parameters should be put before any string() parameters, as in *p/2*.

There can be no more than four parameters of any type (integer() or string()), so the first three parameters has to be integer()'s and the last three string()'s.

p(integer(), integer(), integer(), integer(), string(), string(), string(), string()) -> true | false | error | badarg

Calling this function will trigger the "user" trace probe user_trace_i4s4 in the dyntrace NIF module, sending a trace message containing all the integer()'s and string()'s provided, as well as any user tag set in the current process.

get_tag() -> binary() | undefined

This function returns the user tag set in the current process. If no tag is set or dynamic tracing is not available, it returns `undefined`

get_tag() -> binary() | undefined

This function returns the user tag set in the current process or, if no user tag is present, the last user tag sent to the process together with a message (in the same way as *sequential trace tokens* are spread to other processes together with messages. For an explanation of how user tags can be spread together with messages, see *spread_tag/1*. If no tag is found or dynamic tracing is not available, it returns `undefined`

put_tag(Item) -> binary() | undefined

Types:

`Item = iodata()`

This function sets the user tag of the current process. The user tag is a binary(), but can be specified as any iodata(), which is automatically converted to a binary by this function.

The user tag is provided to the user probes triggered by calls top `dyntrace:p/{1,2,3,4,5,6,7,8}` as well as probes in the efile_driver. In the future, user tags might be added to more probes.

The old user tag (if any) is returned, or `undefined` if no user tag was present or dynamic tracing is not enabled.

spread_tag(boolean()) -> TagData

Types:

`TagData = opaque data that can be used as parameter to` *`restore_tag/1`*

This function controls if user tags are to be spread to other processes with the next message. Spreading of user tags work like spreading of sequential trace tokens, so that a received user tag will be active in the process until the next message arrives (if that message does not also contain the user tag.

This functionality is used when a client process communicates with a file i/o-server to spread the user tag to the I/O-server and then down to the efile_drv driver. By using `spread_tag/1` and `restore_tag/1`, one can enable or disable spreading of user tags to other processes and then restore the previous state of the user tag. The TagData returned from this call contains all previous information so the state (including any previously spread user tags) will be completely restored by a later call to `restore_tag/1`.

The *file* module already spread's tags, so there is no need to manually call these function to get user tags spread to the efile driver through that module.

The most use of this function would be if one for example uses the *io* module to communicate with an I/O-server for a regular file, like in the following example:

```
f() ->
   {ok, F} = file:open("test.tst",[write]),
   Saved = dyntrace:spread_tag(true),
   io:format(F,"Hello world!",[]),
   dyntrace:restore_tag(Saved),
   file:close(F).
```

In this example, any user tag set in the calling process will be spread to the I/O-server when the io:format call is done.

restore_tag(TagData) -> true

Types:

TagData = opaque data returned by *spread_tag/1*

Restores the previous state of user tags and their spreading as it was before a call to *spread_tag/1*. Note that the restoring is not limited to the same process, one can utilize this to turn off spreding in one process and restore it in a newly created, the one that actually is going to send messages:

```
f() ->
    TagData=dyntrace:spread_tag(false),
    spawn(fun() ->
             dyntrace:restore_tag(TagData),
             do_something()
          end),
    do_something_else(),
    dyntrace:restore_tag(TagData).
```

Correctly handling user tags and their spreading might take some effort, as Erlang programs tend to send and receive messages so that sometimes the user tag gets lost due to various things, like double receives or communication with a port (ports do not handle user tags, in the same way as they do not handle regular sequential trace tokens).

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 its 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)

msacc

Erlang module

This module implements some convenience functions for analyzing microstate accounting data. For details about how to use the basic api and what the different states represent see *erlang:statistics(microstate_accounting)*.

Basic Scenario

```
1> msacc:start(1000).
ok
2> msacc:print().
Average thread real-time    : 1000513 us
Accumulated system run-time :    2213 us
Average scheduler run-time  :    1076 us

        Thread      aux check_io emulator       gc    other     port    sleep

Stats per thread:
     async( 0)    0.00%    0.00%    0.00%    0.00%    0.00%    0.00%  100.00%
     async( 1)    0.00%    0.00%    0.00%    0.00%    0.00%    0.00%  100.00%
       aux( 1)    0.00%    0.00%    0.00%    0.00%    0.00%    0.00%   99.99%
 scheduler( 1)    0.00%    0.03%    0.13%    0.00%    0.01%    0.00%   99.82%
 scheduler( 2)    0.00%    0.00%    0.00%    0.00%    0.03%    0.00%   99.97%

Stats per type:
         async    0.00%    0.00%    0.00%    0.00%    0.00%    0.00%  100.00%
           aux    0.00%    0.00%    0.00%    0.00%    0.00%    0.00%   99.99%
     scheduler    0.00%    0.02%    0.06%    0.00%    0.02%    0.00%   99.89%
ok
```

This first command enables microstate accounting for 1000 milliseconds. See *start/0*, *stop/0*, *reset/0* and *start/1* for more details. The second command prints the statistics gathered during that time. First three general statistics are printed.

Average real-time
: The average time spent collecting data in the threads. This should be close to the time which data was collected.

System run-time
: The total run-time of all threads in the system. This is what you get if you call `msacc:stats(total_runtime,Stats).`

Average scheduler run-time
: The average run-time for the schedulers. This is the average amount of time the schedulers did not sleep.

Then one column per state is printed with a the percentage of time this thread spent in the state out of it's own real-time. After the thread specific time, the accumulated time for each type of thread is printed in a similar format.

Since we have the average real-time and the percentage spent in each state we can easily calculate the time spent in each state by multiplying `Average thread real-time` with `Thread state %`, i.e. to get the time Scheduler 1 spent in the emulator state we do `1000513us * 0.13% = 1300us`.

Data Types

```
msacc_data() = [msacc_data_thread()]
```

```
msacc_data_thread() =
    #{'$type' := msacc_data,
      type := msacc_type(),
```

```
      id := msacc_id(),
      counters := msacc_data_counters()}
```

```
msacc_data_counters() = #{msacc_state() => integer() >= 0}
```

A map containing the different microstate accounting states and the number of microseconds spent in it.

```
msacc_stats() = [msacc_stats_thread()]
```

```
msacc_stats_thread() =
    #{'$type' := msacc_stats,
      type := msacc_type(),
      id := msacc_id(),
      system := float(),
      counters := msacc_stats_counters()}
```

A map containing information about a specific thread. The percentages in the map can be either run-time or real-time depending on if `runtime` or `realtime` was requested from *stats/2*. `system` is the percentage of total system time for this specific thread.

```
msacc_stats_counters() =
    #{msacc_state() => #{thread := float(), system := float()}}
```

A map containing the different microstate accounting states. Each value in the map contains another map with the percentage of time that this thread has spent in the specific state. Both the percentage of `system` time and the time for that specific `thread` is part of the map.

```
msacc_type() = scheduler | aux | async
```

```
msacc_id() = integer() >= 0
```

```
msacc_state() =
    alloc |
    aux |
    bif |
    busy_wait |
    check_io |
    emulator |
    ets |
    gc |
    gc_fullsweep |
    nif |
    other |
    port |
    send |
    sleep |
    timers
```

The different states that a thread can be in. See *erlang:statistics(microstate_accounting)* for details.

```
msacc_print_options() = #{system => boolean()}
```

The different options that can be given to *print/2*.

Exports

```
available() -> boolean()
```

This function checks whether microstate accounting is available or not.

`start() -> boolean()`

Start microstate accounting. Returns whether it was previously enabled or disabled.

`start(Time) -> true`

Types:

`Time = timeout()`

Resets all counters and then starts microstate accounting for the given milliseconds.

`stop() -> boolean()`

Stop microstate accounting. Returns whether is was previously enabled or disabled.

`reset() -> boolean()`

Reset microstate accounting counters. Returns whether is was enabled or disabled.

`print() -> ok`

Prints the current microstate accounting to standard out. Same as *`msacc:print(msacc:stats(),#{}).`*

`print(DataOrStats) -> ok`

Types:

`DataOrStats =` ***`msacc_data()`*** `|` ***`msacc_stats()`***

Print the given microstate statistics values to stdout. Same as *`msacc:print(DataOrStats,#{}).`*

`print(DataOrStats, Options) -> ok`

Types:

`DataOrStats =` ***`msacc_data()`*** `|` ***`msacc_stats()`***
`Options =` ***`msacc_print_options()`***

Print the given microstate statistics values to standard out. With many states this can be quite verbose. See the top of this reference manual for a brief description of what the fields mean.

It is possible to print more specific types of statistics by first manipulating the `DataOrStats` using *`stats/2`*. For instance if you want to print the percentage of run-time for each thread you can do:

```
msacc:print(msacc:stats(runtime,msacc:stats())).
```

If you want to only print run-time per thread type you can do:

```
msacc:print(msacc:stats(type,msacc:stats(runtime,msacc:stats()))).
```

Options

`system`
: Print percentage of time spent in each state out of system time as well as thread time. Default: false.

`print(FileOrDevice, DataOrStats, Options) -> ok`

Types:

FileOrDevice = ***file:filename()*** | ***io:device()***
DataOrStats = ***msacc_data()*** | ***msacc_stats()***
Options = ***msacc_print_options()***

Print the given microstate statistics values to the given file or device. The other arguments behave the same way as for `print/2`.

stats() -> ***msacc_data()***

Returns a runtime system independent version of the microstate statistics data presented by `erlang:statistics(microstate_accounting)`. All counters have been normalized to be in microsecond resolution.

stats(Analysis, Stats) -> integer() >= 0

Types:

Analysis = system_realtime | system_runtime
Stats = ***msacc_data()***

Returns the system time for the given microstate statistics values. System time is the accumulated time of all threads.

`realtime`
: Returns all time recorded for all threads.

`runtime`
: Returns all time spent doing work for all threads, i.e. all time not spent in the `sleep` state.

stats(Analysis, Stats) -> ***msacc_stats()***

Types:

Analysis = realtime | runtime
Stats = ***msacc_data()***

Returns fractions of real-time or run-time spent in the various threads from the given microstate statistics values.

stats(Analysis, StatsOrData) -> ***msacc_data()*** | ***msacc_stats()***

Types:

Analysis = type
StatsOrData = ***msacc_data()*** | ***msacc_stats()***

Returns a list of microstate statistics values where the values for all threads of the same type has been merged.

to_file(Filename) -> ok | {error, ***file:posix()***}

Types:

Filename = ***file:name_all()***

Dumps the current microstate statistics counters to a file that can be parsed with *file:consult/1*.

from_file(Filename) -> ***msacc_data()***

Types:

Filename = ***file:name_all()***

Read a file dump produced by *to_file(Filename)*.

scheduler

Erlang module

This module contains utility functions for easy measurement and calculation of scheduler utilization. It act as a wrapper around the more primitive API *erlang:statistics(scheduler_wall_time)*.

The simplest usage is to call the blocking *scheduler:utilization(Seconds)*.

For non blocking and/or continuous calculation of scheduler utilization, the recommended usage is:

- First call *erlang:system_flag(scheduler_wall_time,true)* to enable scheduler wall time measurements.
- Call *get_sample/0* to collect samples with some time in between.
- Call *utilization/2* to calculate the scheduler utilization in the interval between two samples.
- When done call *erlang:system_flag(scheduler_wall_time,false)* to disable scheduler wall time measurements and avoid unecessary cpu overhead.

To get correct values from *utilization/2*, it is important that `scheduler_wall_time` is kept enabled during the entire interval between the two samples. To ensure this, the process that called *erlang:system_flag(scheduler_wall_time,true)* must be kept alive, as `scheduler_wall_time` will automatically be disabled if it terminates.

Data Types

```
sched_sample()
```

```
sched_type() = normal | cpu | io
```

```
sched_id() = integer()
```

```
sched_util_result() =
    [{sched_type(), sched_id(), float(), string()} |
     {total, float(), string()} |
     {weighted, float(), string()}]
```

A list of tuples containing results for individual schedulers as well as aggregated averages. `Util` is the scheduler utilization as a floating point value between 0.0 and 1.0. `Percent` is the same utilization as a more human readable string expressed in percent.

`{normal, SchedulerId, Util, Percent}`
: Scheduler utilization of a normal scheduler with number `SchedulerId`. Schedulers that are not online will also be included. *Online schedulers* have the lowest `SchedulerId`.

`{cpu, SchedulerId, Util, Percent}`
: Scheduler utilization of a dirty-cpu scheduler with number `SchedulerId`.

`{io, SchedulerId, Util, Percent}`
: Scheduler utilization of a dirty-io scheduler with number `SchedulerId`. This tuple will only exist if both samples were taken with *sample_all/0*.

`{total, Util, Percent}`
: Total utilization of all normal and dirty-cpu schedulers.

`{weighted, Util, Percent}`
: Total utilization of all normal and dirty-cpu schedulers, weighted against maximum amount of available CPU time.

Exports

`get_sample() ->` ***`sched_sample()`*** `| undefined`

Returns a scheduler utilization sample for normal and dirty-cpu schedulers. Returns `undefined` if system flag *`scheduler_wall_time`* has not been enabled.

`get_sample_all() ->` ***`sched_sample()`*** `| undefined`

Return a scheduler utilization sample for all schedulers, including dirty-io schedulers. Returns `undefined` if system flag *`scheduler_wall_time`* has not been enabled.

`sample() ->` ***`sched_sample()`***

Return a scheduler utilization sample for normal and dirty-cpu schedulers. Will call *`erlang:system_flag(scheduler_wall_time,true)`* first if not already already enabled.

> **Note:**
>
> This function is **not recommended** as there is no way to detect if `scheduler_wall_time` already was enabled or not. If `scheduler_wall_time` has been disabled between two samples, passing them to *`utilization/2`* will yield invalid results.
>
> Instead use *`get_sample/0`* together with *`erlang:system_flag(scheduler_wall_time,_)`*.

`sample_all() ->` ***`sched_sample()`***

Return a scheduler utilization sample for all schedulers, including dirty-io schedulers. Will call *`erlang:system_flag(scheduler_wall_time,true)`* first if not already already enabled.

> **Note:**
>
> This function is **not recommended** for same reason as `sample/0`. Instead use *`get_sample_all/0`* together with *`erlang:system_flag(scheduler_wall_time,_)`*.

`utilization(Seconds) ->` ***`sched_util_result()`***

Types:

`Seconds = integer() >= 1`

Measure utilization for normal and dirty-cpu schedulers during `Seconds` seconds, and then return the result.

Will automatically first enable and then disable *`scheduler_wall_time`*.

`utilization(Sample) ->` ***`sched_util_result()`***

Types:

`Sample =` ***`sched_sample()`***

Calculate scheduler utilizations for the time interval from when `Sample` was taken and "now". The same as calling `scheduler:utilization(Sample, scheduler:sample_all())`.

> **Note:**
>
> This function is **not recommended** as it's so easy to get invalid results without noticing. In particular do not do this:
>
> ```
> scheduler:utilization(scheduler:sample()). % DO NOT DO THIS!
> ```
>
> The above example takes two samples in rapid succession and calculates the scheduler utilization between them. The resulting values will probably be more misleading than informative.
>
> Instead use *`scheduler:utilization/2`* and call *`get_sample/0`* to get samples with some time in between.

`utilization(Sample1, Sample2) ->` ***`sched_util_result()`***

Types:

`Sample1 = Sample2 =` ***`sched_sample()`***

Calculates scheduler utilizations for the time interval between the two samples obtained from calling *`get_sample/0`* or *`get_sample_all/0`*.

This function itself, does not need *`scheduler_wall_time`* to be enabled. However, for a correct result, `scheduler_wall_time` must have been enabled during the entire interval between the two samples.

system_information

Erlang module

Exports

```
sanity_check() -> ok | {failed, Failures}
```

Types:

```
Application = atom()
ApplicationVersion = string()
MissingRuntimeDependencies =
    {missing_runtime_dependencies,
     ApplicationVersion,
     [ApplicationVersion]}
InvalidApplicationVersion =
    {invalid_application_version, ApplicationVersion}
InvalidAppFile = {invalid_app_file, Application}
Failure =
    MissingRuntimeDependencies |
    InvalidApplicationVersion |
    InvalidAppFile
Failures = [Failure]
```

Performs a sanity check on the system. If no issues were found, `ok` is returned. If issues were found, `{failed, Failures}` is returned. All failures found will be part of the `Failures` list. Currently defined `Failure` elements in the `Failures` list:

`InvalidAppFile`

An application has an invalid `.app` file. The second element identifies the application which has the invalid `.app` file.

`InvalidApplicationVersion`

An application has an invalid application version. The second element identifies the application version that is invalid.

`MissingRuntimeDependencies`

An application is missing *runtime dependencies*. The second element identifies the application (with version) that has missing dependencies. The third element contains the missing dependencies.

Note that this check use application versions that are loaded, or will be loaded when used. You might have application versions that satisfies all dependencies installed in the system, but if those are not loaded this check will fail. The system will of course also fail when used like this. This may happen when you have multiple *branched versions* of the same application installed in the system, but you do not use a *boot script* identifying the correct application version.

Currently the sanity check is limited to verifying runtime dependencies found in the `.app` files of all applications. More checks will be introduced in the future. This implies that the return type **will** change in the future.

> **Note:**
>
> An `ok` return value only means that `sanity_check/0` did not find any issues, **not** that no issues exist.

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

Types:

`FileName = `***`file:name_all()`***

`Reason = `***`file:posix()`***` | badarg | terminated | system_limit`

Writes miscellaneous system information to file. This information will typically be requested by the Erlang/OTP team at Ericsson AB when reporting an issue.