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](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:

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

**process_exit**

• **pid**: string: Process ID. Ex. "<0.130.0>"

• **reason**: string: Exit reason. Ex. "normal"

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

Example:

```erlang
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. "logger"

• **type**: string: "register" | "unregister"

Example:

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

**process_scheduled**

• **pid**: string: Process ID. Ex. "<0.130.0>"

• **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:

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

**port_open**

• **pid**: string: Process ID. Ex. "<0.130.0>"

• **driver**: string: Driver name. Ex. "tcp_inet"

• **port**: string: Port ID. Ex. "#Port<0.1031>"

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

Example:

```erlang
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:

```erlang
port_exit: { cpu_id = 5 }, { port = "#Port<0.1887>", reason = "normal" }
```
1.1 LTTng and Erlang/OTP

**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" }
```

**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", depth = 0 }  
```

**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 }"
```

gcMinorEnd

- **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
1.1 LTTng and Erlang/OTP

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

Example:

```erlang
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:

```erlang
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:

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

**driver_init**

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

Example:

```erlang
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. "tcp_inet"
- *port*: string :: Port ID. Ex. ":Port<0.1031>"

Example:

```erlang
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>"
1.1 LTTng and Erlang/OTP

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

**Example:**
```erlang
driver_output: { cpu_id = 2 }, { pid = "<0.198.0>", port = "#Port<0.3677>", driver = "/bin/sh -s unix:cmd", bytes = 36 }
```

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

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

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

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

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

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

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

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

- **driver**: string :: Driver name. Ex. "tcp_inet"

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

- **pid**: string :: Process ID. Ex. "<0.131.0>"
1.1 LTTng and Erlang/OTP

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

Example:

```erlang
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. "tcp_inet"

Example:

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

driver_process_exit

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

Example:

```erlang
driver_process_exit: { cpu_id = 3 }, { pid = "<0.181.0>", port = "#Port<0.3622>", driver = "tcp_inet" }
```

driver_ready_async

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

Example:

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

driver_call

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

Example:

```erlang
driver_call: { cpu_id = 2 }, { pid = "<0.202.0>", port = "#Port<0.3676>", driver = "caller_drv", command = 0, bytes = 82 }
```

driver_control

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

Example:

```erlang
driver_control: { cpu_id = 3 }, { pid = "<0.32767.8191>", port = "#Port<0.8>", driver = "forker", command = 83, bytes = 82 }
```
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_carriers_size = 1343488, mbc_blocks_size = 285296, sbc_carriers = 0, sbc_carriers_size = 0, sbc_blocks = 0, sbc_blocks_size = 0 }
```

**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_carriers_size = 1343488, mbc_blocks_size = 285296, sbc_carriers = 0, sbc_carriers_size = 0, sbc_blocks = 0, sbc_blocks_size = 0 }
```

**carrier_pool_put**
1.1 LTTng and Erlang/OTP

- **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] [lttng]
Eshell V8.0 (abort with ^G)
1>
```

Load the dytrace module.

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

All tracepoints via dytrace 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 dytrace as tracer backend.

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

Start LTTng tracing.

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

Start the os_mon application in Erlang.
Stop LTTng tracing and view the result.

```bash
$ 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 DTrace probe specifications
Probe specifications can be found in erts/emulator/beam/erlang_dtrace.d, and a few example scripts can be found under lib/runtime_tools/examples/.

1.3 SystemTap and Erlang/OTP

1.3.1 Introduction
SystemTap is DTrace for Linux. In fact Erlang’s SystemTap support is build using SystemTap's DTrace compatibility's layer. For an introduction to Erlang DTrace support read SERL_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
  At 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 therefor 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("*")'
```
1.3 SystemTap and Erlang/OTP

Probes in the dtrace.so NIF library like this:

```bash
# 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:

```bash
# 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)
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:
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 built-in match_spec functions of course is allowed:

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

```erlang
h()  -> ok
```

Gives a list of items for brief online help.

```erlang
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`

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

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

```erlang
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:
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 \texttt{n/1} and \texttt{tracer/3}).

- \texttt{all}
  - All processes and ports in the system as well as all processes and ports created hereafter are to be traced.
- \texttt{processes}
  - All processes in the system as well as all processes created hereafter are to be traced.
- \texttt{ports}
  - All ports in the system as well as all ports created hereafter are to be traced.
- \texttt{new}
  - All processes and ports created after the call is are to be traced.
- \texttt{new\_processes}
  - All processes created after the call is are to be traced.
- \texttt{new\_ports}
  - All ports created after the call is are to be traced.
- \texttt{existing}
  - All existing processes and ports are traced.
- \texttt{existing\_processes}
  - All existing processes are traced.
- \texttt{existing\_ports}
  - All existing ports are traced.
- \texttt{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 \texttt{n/1} or \texttt{tracer/3} function.
- \texttt{integer()}
  - The process \texttt{<0.Item.0>} is traced.
- \texttt{{X, Y, Z}}
  - The process \texttt{<X.Y.Z>} is traced.
- \texttt{string()}
  - If the \texttt{Item} is a string "\texttt{<X.Y.Z>}" as returned from \texttt{pid\_to\_list/1}, the process \texttt{<X.Y.Z>} is traced.

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

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

- \texttt{s (send)}
  - Traces the messages the process or port sends.
- \texttt{r (receive)}
  - Traces the messages the process or port receives.
- \texttt{m (messages)}
  - Traces the messages the process or port receives and sends.
- \texttt{c (call)}
  - Traces global function calls for the process according to the trace patterns set in the system (see \texttt{tp/2}).
- \texttt{p (procs)}
  - Traces process related events to the process.
- \texttt{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() | '_'

Ericsson AB. All Rights Reserved.: Runtime_Tools | 19
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)\Rightarrow \text{dbg:tp((dbg,ltp,0),[[[]],[[]],[\{message, two, arguments\}, \{noexist\}]]))}]\}

\{error,\}
\{error,\}
\{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\((\{Module, Function, Arity\}, MatchSpec) -> \{ok, MatchDesc\} \mid \{error, term()\}\)

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 tpe/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 \{\text{saved, N}\} is however ever returned (for obvious reasons).

\[
\text{ctp}() \\
\text{Same as ctpl(\{\_,'_','_\})}
\]

\[
\text{ctp}(\text{Module}) \\
\text{Same as ctpl(\{\text{Module},'_','_\})}
\]

\[
\text{ctp}(\text{Module, Function}) \\
\text{Same as ctpl(\{\text{Module, Function,'_\})}
\]

\[
\text{ctp}(\text{Module, Function, Arity}) \\
\text{Same as ctpl(\{\text{Module, Function, Arity}\})}
\]

\[
\text{ctp}(\{\text{Module, Function, Arity}\}) \rightarrow \{\text{ok, MatchDesc}\} | \{\text{error, term()}\}
\]

This function works as \( ctp/1 \), but only disables tracing set up with \( tp1/2 \) (not with \( tp/2 \)).

\[
\text{ctpg}() \\
\text{Same as ctpg(\{\_,'_','_\})}
\]

\[
\text{ctpg}(\text{Module}) \\
\text{Same as ctpg(\{\text{Module,'_','_\})}
\]

\[
\text{ctpg}(\text{Module, Function}) \\
\text{Same as ctpg(\{\text{Module, Function,'_\})}
\]

\[
\text{ctpg}(\text{Module, Function, Arity}) \\
\text{Same as ctpg(\{\text{Module, Function, Arity}\})}
\]

\[
\text{ctpg}(\{\text{Module, Function, Arity}\}) \rightarrow \{\text{ok, MatchDesc}\} | \{\text{error, term()}\}
\]

This function works as \( ctp/1 \), but only disables tracing set up with \( tp/2 \) (not with \( tp1/2 \)).

\[
\text{ctpe(Event)} \rightarrow \{\text{ok, MatchDesc}\} | \{\text{error, term()}\}
\]

Types:

\[
\begin{align*}
\text{Event} &= \text{send} | \text{'receive'} \\
\text{MatchDesc} &= \{\text{MatchNum}\} \\
\text{MatchNum} &= \{\text{matched, node()}, 1\} | \{\text{matched, node()}, 0, \text{RPCError}\}
\end{align*}
\]

This function clears match specifications for the specified trace event (\text{send} or \text{'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() \rightarrow 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:

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

\[
[1,2]
\]

dtp() \rightarrow 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.

\[
dtp(N) \rightarrow ok
\]\n
Types:
\[
N = \text{integer ()}
\]

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

\[
wtp(Name) \rightarrow ok \mid \{\text{error, IOError}\}\]

Types:
\[
Name = \text{string ()}
\]
\[
\text{IOError} = \text{term ()}
\]

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.

\[
rtp(Name) \rightarrow ok \mid \{\text{error, Error}\}\]

Types:
\[
Name = \text{string ()}
\]
\[
\text{Error} = \text{term ()}
\]

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.

\[
\text{n(Nodename)} -> \{\text{ok}, \text{Nodename}\} | \{\text{error}, \text{Reason}\}
\]

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 cannot 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.

\[
\text{cn(Nodename)} -> \text{ok}
\]

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.

\[
\text{ln()} -> \text{ok}
\]

Shows the list of traced nodes on the console.

\[
\text{tracer()} -> \{\text{ok, pid()}\} | \{\text{error, already_started}\}
\]

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`.

\[
\text{tracer(Type, Data)} -> \{\text{ok, pid()}\} | \{\text{error, Error}\}
\]

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
(<0.22.0>) <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
  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 and clears all trace flags for all processes and all local trace patterns for all functions. Also shuts down all trace clients and closes all trace ports.
Note that no global trace patterns are affected by this function.

stop_clear() -> ok
Same as stop/0, but also clears all trace patterns on global functions calls.

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

(tiger@durin)84> dbg:c(dbg,get_tracer,[]).
(<0.154.0>) ! {<0.152.0>,{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:
If you set the `call` trace flag, you also have to set a **trace pattern** for the functions you want to trace:

```erlang
(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().
{ok,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().
{ok,<0.116.0>}
(tiger@durin)83> dbg:get_tracer/0 -> {ok,<0.143.0>}
{ok,<0.143.0>}
(tiger@durin)83> flush().
Shell got hello
ok
(tiger@durin)71>
```

**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:
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).
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.
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.

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.

```erlang
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 spreading in one process and restore it in a newly created, the one that actually is going to send messages:

```erlang
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

<table>
<thead>
<tr>
<th>Thread</th>
<th>aux</th>
<th>check_io</th>
<th>emulator</th>
<th>gc</th>
<th>other</th>
<th>port</th>
<th>sleep</th>
</tr>
</thead>
<tbody>
<tr>
<td>async( 0)</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>async( 1)</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>100.00%</td>
</tr>
<tr>
<td>aux( 1)</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>99.99%</td>
</tr>
<tr>
<td>scheduler( 1)</td>
<td>0.00%</td>
<td>0.03%</td>
<td>0.13%</td>
<td>0.00%</td>
<td>0.01%</td>
<td>0.00%</td>
<td>99.82%</td>
</tr>
<tr>
<td>scheduler( 2)</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.03%</td>
<td>0.00%</td>
<td>99.97%</td>
</tr>
</tbody>
</table>

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(),
      }
msacc

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() =
  aux |
  async |
  dirty_cpu_scheduler |
  dirty_io_scheduler |
  poll |
  scheduler

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 it was previously enabled or disabled.

reset() -> boolean()
Reset microstate accounting counters. Returns whether it 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:

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

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

```erlang
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)`.
This module contains utility functions for easier measurement and calculation of scheduler utilization, otherwise obtained from calling the more primitive `statistics(scheduler_wall_time)`.

The simplest usage is to call `scheduler:utilization(Seconds)`.

**Data Types**

- `sched_sample()`: Returns a scheduler utilization sample.
- `sched_type() = normal | cpu | io`: Types of schedulers.
- `sched_id() = integer()`: Identifier for scheduler.
- `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.
  - `{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**

- `sample() -> sched_sample()`: Returns a scheduler utilization sample for normal and dirty-cpu schedulers.
- `sample_all() -> sched_sample()`: Returns a scheduler utilization sample for all schedulers, including dirty-io schedulers.
- `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.
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()).

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 sample/0 or sample_all/0.
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.