1 Kernel User's Guide

1.1 Introduction

1.1.1 Scope

The Kernel application has all the code necessary to run the Erlang runtime system: file servers, code servers, and so on. The Kernel application is the first application started. It is mandatory in the sense that the minimal system based on Erlang/OTP consists of Kernel and STDLIB. Kernel contains the following functional areas:

- Start, stop, supervision, configuration, and distribution of applications
- Code loading
- Logging
- Global name service
- Supervision of Erlang/OTP
- Communication with sockets
- Operating system interface

1.1.2 Prerequisites

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

1.2 Socket Usage

1.2.1 Introduction

The socket interface (module) is basically an "thin" layer on top of the OS socket interface. It is assumed that, unless you have special needs, gen_{tcp|udp|sctp} should be sufficient (when they become available).

Note that just because we have a documented and described option, it does not mean that the OS supports it. So its recommended that the user reads the platform specific documentation for the option used.

Asynchronous calls

Some functions allow for an asynchronous call (accept/2, connect/3, recv/3,4, recvfrom/3,4, recvmsg/2,3,5, send/3,4, sendmsg/3,4 and sendto/4,5). This is achieved by setting the Timeout argument to nowait. For instance, if calling the recv/3 function with Timeout set to nowait (i.e. recv(Sock, 0, nowait)) when there is actually nothing to read, it will return with {select, SelectInfo} (SelectInfo contains the SelectRef). When data eventually arrives a 'select message' will be sent to the caller:

{'$socket', socket(), select, SelectRef}

The caller can then make another call to the recv function and now expect data.

Note that all other users are locked out until the 'current user' has called the function (recv in this case). So either immediately call the function or cancel.

The user must also be prepared to receive an abort message:
1.2 Socket Usage

\{'$socket', \socket(), \abort, \Info\}

If the operation is aborted for whatever reason (e.g. if the socket is closed "by someone else"). The \Info\ part contains the abort reason (in this case that the socket has been closed \Info = \{\SelectRef, \closed\}).

The general form of the 'socket' message is:

\{'$socket', \Sock :: \socket(), \Tag :: \atom(), \Info :: \term()\}

Where the format of Info is a function of Tag:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Info value type</th>
</tr>
</thead>
<tbody>
<tr>
<td>select</td>
<td>select_ref()</td>
</tr>
<tr>
<td>abort</td>
<td>[select_ref(), Reason :: \term()]</td>
</tr>
</tbody>
</table>

Table 2.1: socket message info value type

The \select_ref() \ is the same as was received in the SelectInfo.

1.2.2 Socket Options

Options for level \otp:

<table>
<thead>
<tr>
<th>Option Name</th>
<th>Value Type</th>
<th>Set</th>
<th>Get</th>
<th>Other Requirements and comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>assoc_id</td>
<td>integer()</td>
<td>no</td>
<td>yes</td>
<td>type = seqpacket, protocol = sctp, is an association</td>
</tr>
<tr>
<td>debug</td>
<td>boolean()</td>
<td>yes</td>
<td>yes</td>
<td>none</td>
</tr>
<tr>
<td>iow</td>
<td>boolean()</td>
<td>yes</td>
<td>yes</td>
<td>none</td>
</tr>
<tr>
<td>controlling_process</td>
<td>pid()</td>
<td>yes</td>
<td>yes</td>
<td>none</td>
</tr>
<tr>
<td>rcvbuf</td>
<td>default</td>
<td>pos_integer()</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>rcvctrlbuf</td>
<td>default</td>
<td>pos_integer()</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>sndctrlbuf</td>
<td>default</td>
<td>pos_integer()</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>fd</td>
<td>integer()</td>
<td>no</td>
<td>yes</td>
<td>none</td>
</tr>
</tbody>
</table>

Table 2.2: option levels
## 1.2 Socket Usage

Options for level *socket*:

<table>
<thead>
<tr>
<th>Option Name</th>
<th>Value Type</th>
<th>Set</th>
<th>Get</th>
<th>Other Requirements and comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>acceptconn</td>
<td>boolean()</td>
<td>no</td>
<td>yes</td>
<td>none</td>
</tr>
<tr>
<td>bindtodevice</td>
<td>string()</td>
<td>yes</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>broadcast</td>
<td>boolean()</td>
<td>yes</td>
<td>yes</td>
<td>type = dgram</td>
</tr>
<tr>
<td>debug</td>
<td>integer()</td>
<td>yes</td>
<td>yes</td>
<td>may require admin capability</td>
</tr>
<tr>
<td>domain</td>
<td>domain()</td>
<td>no</td>
<td>yes</td>
<td>Not on FreeBSD (for instance)</td>
</tr>
<tr>
<td>dontroute</td>
<td>boolean()</td>
<td>yes</td>
<td>yes</td>
<td>none</td>
</tr>
<tr>
<td>keepalive</td>
<td>boolean()</td>
<td>yes</td>
<td>yes</td>
<td>none</td>
</tr>
<tr>
<td>linger</td>
<td>abort</td>
<td>linger()</td>
<td>yes</td>
<td>none</td>
</tr>
<tr>
<td>oobinline</td>
<td>boolean()</td>
<td>yes</td>
<td>yes</td>
<td>none</td>
</tr>
<tr>
<td>peek_off</td>
<td>integer()</td>
<td>yes</td>
<td>yes</td>
<td>domain = local (unix). Currently disabled due to a possible infinite loop when calling recv([peek]) the second time.</td>
</tr>
<tr>
<td>priority</td>
<td>integer()</td>
<td>yes</td>
<td>yes</td>
<td>none</td>
</tr>
<tr>
<td>protocol</td>
<td>protocol()</td>
<td>no</td>
<td>yes</td>
<td>Not on (some) Darwin (for instance)</td>
</tr>
<tr>
<td>rcvbuf</td>
<td>non_neg_integer()</td>
<td>yes</td>
<td>yes</td>
<td>none</td>
</tr>
<tr>
<td>rcvlowat</td>
<td>non_neg_integer()</td>
<td>yes</td>
<td>yes</td>
<td>none</td>
</tr>
<tr>
<td>rcvtimeo</td>
<td>timeval()</td>
<td>yes</td>
<td>yes</td>
<td>This option is not normally</td>
</tr>
</tbody>
</table>
supported (see why below). OTP has to be explicitly built with the --enable-esock-rcvsndtime configure option for this to be available. Since our implementation is **nonblocking**, it is unknown if and how this option works, or even if it may cause malfunctions. Therefore, we do not recommend setting this option. Instead, use the **Timeout** argument to, for instance, the recv/3 function.

<table>
<thead>
<tr>
<th>Option</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reuseaddr</td>
<td>boolean()</td>
<td>yes</td>
<td>none</td>
</tr>
<tr>
<td>reuseport</td>
<td>boolean()</td>
<td>yes</td>
<td>domain = inet</td>
</tr>
<tr>
<td>sndbuf</td>
<td>non_neg_integer()</td>
<td>yes</td>
<td>none</td>
</tr>
<tr>
<td>sndlowat</td>
<td>non_neg_integer()</td>
<td>yes</td>
<td>not changeable on Linux</td>
</tr>
<tr>
<td>sndtimeo</td>
<td>timeval()</td>
<td>yes</td>
<td>This option is not normally supported (see why below). OTP has to be explicitly built with the --enable-esock-rcvsndtime configure option for this to be available. Since our implementation is <strong>nonblocking</strong>, it is unknown if and how this option works, or even if it may cause malfunctions. Therefore, we do not recommend setting this option.</td>
</tr>
</tbody>
</table>
1.2 Socket Usage

Instead, use the Timeout argument to, for instance, the send/3 function.

<table>
<thead>
<tr>
<th>Option Name</th>
<th>Value Type</th>
<th>Set</th>
<th>Get</th>
<th>Other Requirements and comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>timestamp</td>
<td>boolean()</td>
<td>yes</td>
<td>yes</td>
<td>none</td>
</tr>
<tr>
<td>type</td>
<td>type()</td>
<td>no</td>
<td>yes</td>
<td>none</td>
</tr>
</tbody>
</table>

Table 2.3: socket options

Options for level ip:

<table>
<thead>
<tr>
<th>Option Name</th>
<th>Value Type</th>
<th>Set</th>
<th>Get</th>
<th>Other Requirements and comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_membership</td>
<td>ip_mreq()</td>
<td>yes</td>
<td>no</td>
<td>none</td>
</tr>
<tr>
<td>add_source_membership</td>
<td>ip_mreq_source()</td>
<td>yes</td>
<td>no</td>
<td>none</td>
</tr>
<tr>
<td>block_source</td>
<td>ip_mreq_source()</td>
<td>yes</td>
<td>no</td>
<td>none</td>
</tr>
<tr>
<td>drop_membership</td>
<td>ip_mreq()</td>
<td>yes</td>
<td>no</td>
<td>none</td>
</tr>
<tr>
<td>drop_source_membership</td>
<td>ip_mreq_source()</td>
<td>yes</td>
<td>no</td>
<td>none</td>
</tr>
<tr>
<td>freebind</td>
<td>boolean()</td>
<td>yes</td>
<td>yes</td>
<td>none</td>
</tr>
<tr>
<td>hdrincl</td>
<td>boolean()</td>
<td>yes</td>
<td>yes</td>
<td>type = raw</td>
</tr>
<tr>
<td>minttl</td>
<td>integer()</td>
<td>yes</td>
<td>yes</td>
<td>type = raw</td>
</tr>
<tr>
<td>msfilter</td>
<td>null</td>
<td>ip_msfilter()</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>mtu</td>
<td>integer()</td>
<td>no</td>
<td>yes</td>
<td>type = raw</td>
</tr>
<tr>
<td>mtu_discover</td>
<td>ip_pmtudisc()</td>
<td>yes</td>
<td>yes</td>
<td>none</td>
</tr>
<tr>
<td>multicast_all</td>
<td>boolean()</td>
<td>yes</td>
<td>yes</td>
<td>none</td>
</tr>
<tr>
<td>multicast_if</td>
<td>any</td>
<td>ip4_address()</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>multicast_loop</td>
<td>boolean()</td>
<td>yes</td>
<td>yes</td>
<td>none</td>
</tr>
<tr>
<td>multicast_ttl</td>
<td>uint8()</td>
<td>yes</td>
<td>yes</td>
<td>none</td>
</tr>
<tr>
<td>nodefrag</td>
<td>boolean()</td>
<td>yes</td>
<td>yes</td>
<td>type = raw</td>
</tr>
<tr>
<td>pktinfo</td>
<td>boolean()</td>
<td>yes</td>
<td>yes</td>
<td>type = dgram</td>
</tr>
<tr>
<td>recvdstaddr</td>
<td>boolean()</td>
<td>yes</td>
<td>yes</td>
<td>type = dgram</td>
</tr>
</tbody>
</table>
Table 2.4: ip options

Options for level ipv6:

<table>
<thead>
<tr>
<th>Option Name</th>
<th>Value Type</th>
<th>Set</th>
<th>Get</th>
<th>Other Requirements and comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>addrform</td>
<td>inet</td>
<td>yes</td>
<td>no</td>
<td>allowed only for IPv6 sockets that are connected and bound to a v4-mapped-on-v6 address</td>
</tr>
<tr>
<td>add_membership</td>
<td>ipv6_mreq()</td>
<td>yes</td>
<td>no</td>
<td>none</td>
</tr>
<tr>
<td>authhdr</td>
<td>boolean()</td>
<td>yes</td>
<td>yes</td>
<td>type = dgram</td>
</tr>
<tr>
<td>drop_membership</td>
<td>ipv6_mreq()</td>
<td>yes</td>
<td>no</td>
<td>none</td>
</tr>
<tr>
<td>dstopts</td>
<td>boolean()</td>
<td>yes</td>
<td>yes</td>
<td>type = dgram</td>
</tr>
<tr>
<td>Variable</td>
<td>Type</td>
<td>Default</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------</td>
<td>---------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>flowinfo</td>
<td>boolean()</td>
<td>yes</td>
<td>type = dgram</td>
<td>raw, requires superuser privileges to update</td>
</tr>
<tr>
<td>hoplimit</td>
<td>boolean()</td>
<td>yes</td>
<td>type = dgram</td>
<td>raw. On some platforms (e.g. FreeBSD) it is used to set in order to get hoplimit as a control message header. On others (e.g. Linux), recvhoplimit is set in order to get hoplimit.</td>
</tr>
<tr>
<td>hopopts</td>
<td>boolean()</td>
<td>yes</td>
<td>type = dgram</td>
<td>raw. requires superuser privileges to update</td>
</tr>
<tr>
<td>mtu</td>
<td>boolean()</td>
<td>yes</td>
<td>Get: Only after the socket has been connected</td>
<td></td>
</tr>
<tr>
<td>mtu_discover</td>
<td>ipv6_pmtudisc()</td>
<td>yes</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>multicast_hops</td>
<td>default</td>
<td>uint8()</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>multicast_if</td>
<td>integer()</td>
<td>yes</td>
<td>type = dgram</td>
<td>raw</td>
</tr>
<tr>
<td>multicast_loop</td>
<td>boolean()</td>
<td>yes</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>recverr</td>
<td>boolean()</td>
<td>yes</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>recvhoplimit</td>
<td>boolean()</td>
<td>yes</td>
<td>type = dgram</td>
<td>raw. On some platforms (e.g. Linux), recvhoplimit is set in order to get hoplimit</td>
</tr>
<tr>
<td>recvpktinfo</td>
<td>pktinfo</td>
<td>boolean()</td>
<td>yes</td>
<td>type = dgram</td>
</tr>
</tbody>
</table>
### 1.2 Socket Usage

**Table 2.5: ipv6 options**

<table>
<thead>
<tr>
<th>Option Name</th>
<th>Value Type</th>
<th>Set</th>
<th>Get</th>
<th>Other Requirements and comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>recvclass</td>
<td>boolean()</td>
<td>yes</td>
<td>yes</td>
<td>type = dgram</td>
</tr>
<tr>
<td>router_alert</td>
<td>integer()</td>
<td>yes</td>
<td>yes</td>
<td>type = raw</td>
</tr>
<tr>
<td>rthdr</td>
<td>boolean()</td>
<td>yes</td>
<td>yes</td>
<td>type = dgram</td>
</tr>
<tr>
<td>tclass</td>
<td>boolean()</td>
<td>yes</td>
<td>yes</td>
<td>type = dgram</td>
</tr>
<tr>
<td>unicast_hops</td>
<td>default</td>
<td>uint8()</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>v6only</td>
<td>boolean()</td>
<td>yes</td>
<td>no</td>
<td>none</td>
</tr>
</tbody>
</table>

Options for level tcp:

**Table 2.6: tcp options**

<table>
<thead>
<tr>
<th>Option Name</th>
<th>Value Type</th>
<th>Set</th>
<th>Get</th>
<th>Other Requirements and comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>congestion</td>
<td>string()</td>
<td>yes</td>
<td>yes</td>
<td>none</td>
</tr>
<tr>
<td>maxseg</td>
<td>integer()</td>
<td>yes</td>
<td>yes</td>
<td>Set not allowed on all platforms.</td>
</tr>
<tr>
<td>nodelay</td>
<td>boolean()</td>
<td>yes</td>
<td>yes</td>
<td>none</td>
</tr>
</tbody>
</table>
1.3 Logging

Options for level udp:

<table>
<thead>
<tr>
<th>Option Name</th>
<th>Value Type</th>
<th>Set</th>
<th>Get</th>
<th>Other Requirements and comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>cork</td>
<td>boolean()</td>
<td>yes</td>
<td>yes</td>
<td>none</td>
</tr>
</tbody>
</table>

Table 2.7: udp options

Options for level sctp:

<table>
<thead>
<tr>
<th>Option Name</th>
<th>Value Type</th>
<th>Set</th>
<th>Get</th>
<th>Other Requirements and comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>associnfo</td>
<td>sctp_assocparams()</td>
<td>yes</td>
<td>yes</td>
<td>none</td>
</tr>
<tr>
<td>autoclose</td>
<td>non_neg_integer()</td>
<td>yes</td>
<td>yes</td>
<td>none</td>
</tr>
<tr>
<td>disable.fragments</td>
<td>boolean()</td>
<td>yes</td>
<td>yes</td>
<td>none</td>
</tr>
<tr>
<td>events</td>
<td>sctp_event_subscribe()</td>
<td>yes</td>
<td>no</td>
<td>none</td>
</tr>
<tr>
<td>initmsg</td>
<td>sctp_initmsg()</td>
<td>yes</td>
<td>yes</td>
<td>none</td>
</tr>
<tr>
<td>maxseg</td>
<td>non_neg_integer()</td>
<td>yes</td>
<td>yes</td>
<td>none</td>
</tr>
<tr>
<td>nodelay</td>
<td>boolean()</td>
<td>yes</td>
<td>yes</td>
<td>none</td>
</tr>
<tr>
<td>rtoinfo</td>
<td>sctp_rtoinfo()</td>
<td>yes</td>
<td>yes</td>
<td>none</td>
</tr>
</tbody>
</table>

Table 2.8: sctp options

1.3 Logging

Erlang/OTP 21.0 provides a standard API for logging through Logger, which is part of the Kernel application. Logger consists of the API for issuing log events, and a customizable backend where log handlers, filters and formatters can be plugged in.

By default, the Kernel application installs one log handler at system start. This handler is named default. It receives and processes standard log events produced by the Erlang runtime system, standard behaviours and different Erlang/OTP applications. The log events are by default written to the terminal.

You can also configure the system so that the default handler prints log events to a single file, or to a set of wrap logs via disk_log.

By configuration, you can also modify or disable the default handler, replace it by a custom handler, and install additional handlers.
1.3 Logging

Note:
Since Logger is new in Erlang/OTP 21.0, we do reserve the right to introduce changes to the Logger API and functionality in patches following this release. These changes might or might not be backwards compatible with the initial version.

1.3.1 Overview

A log event consists of a log level, the message to be logged, and metadata.

The Logger backend forwards log events from the API, first through a set of primary filters, then through a set of secondary filters attached to each log handler. The secondary filters are in the following named handler filters.

Each filter set consists of a log level check, followed by zero or more filter functions.

The following figure shows a conceptual overview of Logger. The figure shows two log handlers, but any number of handlers can be installed.
Log levels are expressed as atoms. Internally in Logger, the atoms are mapped to integer values, and a log event passes the log level check if the integer value of its log level is less than or equal to the currently configured log level. That is, the check passes if the event is equally or more severe than the configured level. See section Log Level for a listing and description of all log levels.

The primary log level can be overridden by a log level configured per module. This is to, for instance, allow more verbose logging from a specific part of the system.

Filter functions can be used for more sophisticated filtering than the log level check provides. A filter function can stop or pass a log event, based on any of the event’s contents. It can also modify all parts of the log event. See section Filters for more details.

If a log event passes through all primary filters and all handler filters for a specific handler, Logger forwards the event to the **handler callback**. The handler formats and prints the event to its destination. See section Handlers for more details.
Everything up to and including the call to the handler callbacks is executed on the client process, that is, the process where the log event was issued. It is up to the handler implementation if other processes are involved or not.

The handlers are called in sequence, and the order is not defined.

### 1.3.2 Logger API

The API for logging consists of a set of macros, and a set of functions on the form `logger:Level/1,2,3`, which are all shortcuts for `logger:log(Level,Arg1[,Arg2[,Arg3]])`.

The macros are defined in `logger.hrl`, which is included in a module with the directive

```
-include_lib("kernel/include/logger.hrl").
```

The difference between using the macros and the exported functions is that macros add location (originator) information to the metadata, and performs lazy evaluation by wrapping the logger call in a case statement, so it is only evaluated if the log level of the event passes the primary log level check.

### Log Level

The log level indicates the severity of a event. In accordance with the Syslog protocol, **RFC 5424**, eight log levels can be specified. The following table lists all possible log levels by name (atom), integer value, and description:

<table>
<thead>
<tr>
<th>Level</th>
<th>Integer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>emergency</td>
<td>0</td>
<td>system is unusable</td>
</tr>
<tr>
<td>alert</td>
<td>1</td>
<td>action must be taken immediately</td>
</tr>
<tr>
<td>critical</td>
<td>2</td>
<td>critical conditions</td>
</tr>
<tr>
<td>error</td>
<td>3</td>
<td>error conditions</td>
</tr>
<tr>
<td>warning</td>
<td>4</td>
<td>warning conditions</td>
</tr>
<tr>
<td>notice</td>
<td>5</td>
<td>normal but significant conditions</td>
</tr>
<tr>
<td>info</td>
<td>6</td>
<td>informational messages</td>
</tr>
<tr>
<td>debug</td>
<td>7</td>
<td>debug-level messages</td>
</tr>
</tbody>
</table>

Table 3.1: Log Levels

Notice that the integer value is only used internally in Logger. In the API, you must always use the atom. To compare the severity of two log levels, use `logger:compare_levels/2`.

### Log Message

The log message contains the information to be logged. The message can consist of a format string and arguments (given as two separate parameters in the Logger API), a string or a report. The latter, which is either a map or a key-value list, can be accompanied by a **report callback** specified in the log event's metadata. The report callback is a convenience function that the formatter can use to convert the report to a format string and arguments, or directly to a string. The formatter can also use its own conversion function, if no callback is provided, or if a customized formatting is desired.
1.3 Logging

The report callback must be a fun with one or two arguments. If it takes one argument, this is the report itself, and the fun returns a format string and arguments:

```erlang
fun((logger:report()) -> {io:format(),[term()]})
```

If it takes two arguments, the first is the report, and the second is a map containing extra data that allows direct conversion to a string:

```erlang
fun((logger:report(),logger:report_cb_config()) -> unicode:chardata())
```

The fun must obey the `depth` and `chars_limit` parameters provided in the second argument, as the formatter cannot do anything useful of these parameters with the returned string. The extra data also contains a field named `single_line`, indicating if the printed log message may contain line breaks or not. This variant is used when the formatting of the report depends on the size or single line parameters.

Example, format string and arguments:

```erlang
logger:error("The file does not exist: ~ts",[Filename])
```

Example, string:

```erlang
logger:notice("Something strange happened!"
```

Example, report, and metadata with report callback:

```erlang
logger:debug(#{got => connection_request, id => Id, state => State},
#{report_cb => fun(R) -> {"~p",[R]} end})
```

The log message can also be provided through a fun for lazy evaluation. The fun is only evaluated if the primary log level check passes, and is therefore recommended if it is expensive to generate the message. The lazy fun must return a string, a report, or a tuple with format string and arguments.

**Metadata**

Metadata contains additional data associated with a log message. Logger inserts some metadata fields by default, and the client can add custom metadata in two different ways:

Set process metadata

Process metadata is set and updated with `logger:set_process_metadata/1` and `logger:update_process_metadata/1`, respectively. This metadata applies to the process on which these calls are made, and Logger adds the metadata to all log events issued on that process.

Add metadata to a specific log event

Metadata associated with one specific log event is given as the last parameter to the log macro or Logger API function when the event is issued. For example:

```erlang
?LOG_ERROR("Connection closed",#{context => server})
```

See the description of the `logger:metadata()` type for information about which default keys Logger inserts, and how the different metadata maps are merged.

1.3.3 Filters

Filters can be primary, or attached to a specific handler. Logger calls the primary filters first, and if they all pass, it calls the handler filters for each handler. Logger calls the handler callback only if all filters attached to the handler in question also pass.

A filter is defined as:
where \texttt{FilterFun} is a function of arity 2, and \texttt{Extra} is any term. When applying the filter, Logger calls the function with the log event as the first argument, and the value of \texttt{Extra} as the second argument. See \texttt{logger:filter()} for type definitions.

The filter function can return \texttt{stop}, \texttt{ignore} or the (possibly modified) log event.

If \texttt{stop} is returned, the log event is immediately discarded. If the filter is primary, no handler filters or callbacks are called. If it is a handler filter, the corresponding handler callback is not called, but the log event is forwarded to filters attached to the next handler, if any.

If the log event is returned, the next filter function is called with the returned value as the first argument. That is, if a filter function modifies the log event, the next filter function receives the modified event. The value returned from the last filter function is the value that the handler callback receives.

If the filter function returns \texttt{ignore}, it means that it did not recognize the log event, and thus leaves to other filters to decide the event's destiny.

The configuration option \texttt{filter_default} specifies the behaviour if all filter functions return \texttt{ignore}, or if no filters exist. \texttt{filter_default} is by default set to \texttt{log}, meaning that if all existing filters ignore a log event, Logger forwards the event to the handler callback. If \texttt{filter_default} is set to \texttt{stop}, Logger discards such events.

Primary filters are added with \texttt{logger:add_primary_filter/2} and removed with \texttt{logger:remove_primary_filter/1}. They can also be added at system start via the Kernel configuration parameter \texttt{logger}.

Handler filters are added with \texttt{logger:add_handler_filter/3} and removed with \texttt{logger:remove_handler_filter/2}. They can also be specified directly in the configuration when adding a handler with \texttt{logger:add_handler/3} or via the Kernel configuration parameter \texttt{logger}.

To see which filters are currently installed in the system, use \texttt{logger:get_config/0}, or \texttt{logger:get_primary_config/0} and \texttt{logger:get_handler_config/1}. Filters are listed in the order they are applied, that is, the first filter in the list is applied first, and so on.

For convenience, the following built-in filters exist:

\texttt{logger_filters:domain/2}

Provides a way of filtering log events based on a \texttt{domain} field in Metadata.

\texttt{logger_filters:level/2}

Provides a way of filtering log events based on the log level.

\texttt{logger_filters:progress/2}

Stops or allows progress reports from \texttt{supervisor} and \texttt{application_controller}.

\texttt{logger_filters:remote_gl/2}

Stops or allows log events originating from a process that has its group leader on a remote node.

\subsection{1.3.4 Handlers}

A handler is defined as a module exporting at least the following callback function:

\begin{verbatim}
log(LogEvent, Config) -> void()
\end{verbatim}

This function is called when a log event has passed through all primary filters, and all handler filters attached to the handler in question. The function call is executed on the client process, and it is up to the handler implementation if other processes are involved or not.
1.3 Logging

Logger allows adding multiple instances of a handler callback. That is, if a callback module implementation allows it, you can add multiple handler instances using the same callback module. The different instances are identified by unique handler identities.

In addition to the mandatory callback function `log/2`, a handler module can export the optional callback functions `adding_handler/1`, `changing_config/3`, `filter_config/1`, and `removing_handler/1`. See section Handler Callback Functions in the `logger(3)` manual page for more information about these function.

The following built-in handlers exist:

- `logger_std_h`
  
  This is the default handler used by OTP. Multiple instances can be started, and each instance will write log events to a given destination, terminal or file.

- `logger_disk_log_h`
  
  This handler behaves much like `logger_std_h`, except it uses `disk_log` as its destination.

- `error_logger`
  
  This handler is provided for backwards compatibility only. It is not started by default, but will be automatically started the first time an `error_logger` event handler is added with `error_logger:add_report_handler/1,2`.

  The old `error_logger` event handlers in STDLIB and SASL still exist, but they are not added by Erlang/OTP 21.0 or later.

1.3.5 Formatters

A formatter can be used by the handler implementation to do the final formatting of a log event, before printing to the handler's destination. The handler callback receives the formatter information as part of the handler configuration, which is passed as the second argument to `HModule:log/2`.

The formatter information consist of a formatter module, `FModule` and its configuration, `FConfig`. `FModule` must export the following function, which can be called by the handler:

```
format(LogEvent,FConfig) -> FormattedLogEntry
```

The formatter information for a handler is set as a part of its configuration when the handler is added. It can also be changed during runtime with `logger:set_handler_config(HandlerId,formatter, {FModule,FConfig})`, which overwrites the current formatter information, or with `logger:update_formatter_config/2,3`, which only modifies the formatter configuration.

If the formatter module exports the optional callback function `check_config(FConfig)`, Logger calls this function when the formatter information is set or modified, to verify the validity of the formatter configuration.

If no formatter information is specified for a handler, Logger uses `logger_formatter` as default. See the `logger_formatter(3)` manual page for more information about this module.

1.3.6 Configuration

At system start, Logger is configured through Kernel configuration parameters. The parameters that apply to Logger are described in section Kernel Configuration Parameters. Examples are found in section Configuration Examples.

During runtime, Logger configuration is changed via API functions. See section Configuration API Functions in the `logger(3)` manual page.

Primary Logger Configuration

Logger API functions that apply to the primary Logger configuration are:
The primary Logger configuration is a map with the following keys:

- **level** = \( \text{logger:level()} \mid \text{all} \mid \text{none} \)
  
  Specifies the primary log level, that is, log event that are equally or more severe than this level, are forwarded to the primary filters. Less severe log events are immediately discarded.

  See section Log Level for a listing and description of possible log levels.

  The initial value of this option is set by the Kernel configuration parameter \texttt{logger_level}. It is changed during runtime with \texttt{logger:set_primary_config(level,Level)}.

  Defaults to \texttt{notice}.

- **filters** = \{[FilterId,Filter]\}
  
  Specifies the primary filters.

  - FilterId = \text{logger:filter_id()}
  - Filter = \text{logger:filter()}

  The initial value of this option is set by the Kernel configuration parameter \texttt{logger}. During runtime, primary filters are added and removed with \texttt{logger:add_primary_filter/2} and \texttt{logger:remove_primary_filter/1}, respectively.

  See section Filters for more detailed information.

  Defaults to \[\].

- **filter_default** = \text{log} \mid \text{stop}
  
  Specifies what happens to a log event if all filters return \texttt{ignore}, or if no filters exist.

  See section Filters for more information about how this option is used.

  Defaults to \texttt{log}.

### Handler Configuration

Logger API functions that apply to handler configuration are:

- \text{get_handler_config/0,1}
- \text{set_handler_config/2,3}
- \text{update_handler_config/2,3}
- \text{add_handler_filter/3}
- \text{remove_handler_filter/2}
- \text{update_formatter_config/2,3}

The configuration for a handler is a map with the following keys:

- **id** = \text{logger:handler_id()}
  
  Automatically inserted by Logger. The value is the same as the HandlerId specified when adding the handler, and it cannot be changed.
1.3 Logging

module = module()

Automatically inserted by Logger. The value is the same as the Module specified when adding the handler, and it cannot be changed.

level = logger:level() | all | none

Specifies the log level for the handler, that is, log events that are equally or more severe than this level, are forwarded to the handler filters for this handler.

See section Log Level for a listing and description of possible log levels.

The log level is specified when adding the handler, or changed during runtime with, for instance, logger:set_handler_config(HandlerId,level,Level).

Defaults to all.

filters = [{FilterId,Filter}]

Specifies the handler filters.

- FilterId = logger:filter_id()
- Filter = logger:filter()

Handler filters are specified when adding the handler, or added or removed during runtime with logger:add_handler_filter/3 and logger:remove_handler_filter/2, respectively.

See Filters for more detailed information.

Defaults to [].

filter_default = log | stop

Specifies what happens to a log event if all filters return ignore, or if no filters exist.

See section Filters for more information about how this option is used.

Defaults to log.

formatter = {FormatterModule,FormatterConfig}

Specifies a formatter that the handler can use for converting the log event term to a printable string.

- FormatterModule = module()
- FormatterConfig = logger:formatter_config()

The formatter information is specified when adding the handler. The formatter configuration can be changed during runtime with logger:update_formatter_config/2,3, or the complete formatter information can be overwritten with, for instance, logger:set_handler_config/3.

See section Formatters for more detailed information.

Defaults to {logger_formatter,DefaultFormatterConfig}. See the logger_formatter(3) manual page for information about this formatter and its default configuration.

config = term()

Handler specific configuration, that is, configuration data related to a specific handler implementation.

The configuration for the built-in handlers is described in the logger_std_h(3) and logger_disk_log_h(3) manual pages.

Notice that level and filters are obeyed by Logger itself before forwarding the log events to each handler, while formatter and all handler specific options are left to the handler implementation.

Kernel Configuration Parameters

The following Kernel configuration parameters apply to Logger:
logger = Config

Specifies the configuration for Logger, except the primary log level, which is specified with logger_level, and the compatibility with SASL Error Logging, which is specified with logger_sasl_compatible.

With this parameter, you can modify or disable the default handler, add custom handlers and primary logger filters, set log levels per module, and modify the proxy configuration.

Config is any (zero or more) of the following:

{handler, default, undefined}

Disables the default handler. This allows another application to add its own default handler.

Only one entry of this type is allowed.

{handler, HandlerId, Module, HandlerConfig}

If HandlerId is default, then this entry modifies the default handler, equivalent to calling

```lua
logger:remove_handler(default)
```

followed by

```lua
logger:add_handler(default, Module, HandlerConfig)
```

For all other values of HandlerId, this entry adds a new handler, equivalent to calling

```lua
logger:add_handler(HandlerId, Module, HandlerConfig)
```

Multiple entries of this type are allowed.

{filters, FilterDefault, [Filter]}

Adds the specified primary filters.

- FilterDefault = log | stop
- Filter = {FilterId, {FilterFun, FilterConfig}}

Equivalent to calling

```lua
logger:add_primary_filter(FilterId, {FilterFun, FilterConfig})
```

for each Filter.

FilterDefault specifies the behaviour if all primary filters return ignore, see section Filters.

Only one entry of this type is allowed.

{module_level, Level, [Module]}

Sets module log level for the given modules. Equivalent to calling

```lua
logger:set_module_level(Module, Level)
```

for each Module.

Multiple entries of this type are allowed.
1.3 Logging

(proxy, ProxyConfig)

Sets the proxy configuration, equivalent to calling

```
logger:set_proxy_config(ProxyConfig)
```

Only one entry of this type is allowed.

See section Configuration Examples for examples using the logger parameter for system configuration.

**logger_level = Level**

Specifies the primary log level. See the `kernel(6)` manual page for more information about this parameter.

**logger_sasl_compatible = true | false**

Specifies Logger's compatibility with SASL Error Logging. See the `kernel(6)` manual page for more information about this parameter.

**Configuration Examples**

The value of the Kernel configuration parameter `logger` is a list of tuples. It is possible to write the term on the command line when starting an erlang node, but as the term grows, a better approach is to use the system configuration file. See the `config(4)` manual page for more information about this file.

Each of the following examples shows a simple system configuration file that configures Logger according to the description.

Modify the default handler to print to a file instead of `standard_io`:

```
[{kernel,
  [{logger,
    [{handler, default, logger_std_h, % {handler, HandlerId, Module,
      #{config => #{file => "log/erlang.log"}}} % Config}
    ]}]
].
```

Modify the default handler to print each log event as a single line:

```
[{kernel,
  [{logger,
    [{handler, default, logger_std_h,
      #{formatter => {logger_formatter, #{single_line => true}}}}
    ]}]
].
```

Modify the default handler to print the pid of the logging process for each log event:

```
[{kernel,
  [{logger,
    [{handler, default, logger_std_h,
      #{formatter => {logger_formatter,
        #{template => [time, ",pid," ",msg,\n"}}} % Config}
    ]}]
].
```

Modify the default handler to only print errors and more severe log events to "log/erlang.log", and add another handler to print all log events to "log/debug.log". 
1.3.7 Backwards Compatibility with error_logger

Logger provides backwards compatibility with error_logger in the following ways:

API for Logging

The error_logger API still exists, but should only be used by legacy code. It will be removed in a later release.

Calls to error_logger:error_report/1,2, error_logger:error_msg/1,2, and corresponding functions for warning and info messages, are all forwarded to Logger as calls to logger:log(Level,Report,Metadata).

Level = error | warning | info and is taken from the function name. Report contains the actual log message, and Metadata contains additional information which can be used for creating backwards compatible events for legacy error_logger event handlers, see section Legacy Event Handlers.

Output Format

To get log events on the same format as produced by error_logger_tty_h and error_logger_file_h, use the default formatter, logger_formatter, with configuration parameter legacy_header set to true. This is the default configuration of the default handler started by Kernel.

Default Format of Log Events from OTP

By default, all log events originating from within OTP, except the former so called "SASL reports", look the same as before.

SASL Reports

By SASL reports we mean supervisor reports, crash reports and progress reports.

Prior to Erlang/OTP 21.0, these reports were only logged when the SASL application was running, and they were printed through SASL's own event handlers sasl_report_tty_h and sasl_report_file_h.

The destination of these log events was configured by SASL configuration parameters.

Due to the specific event handlers, the output format slightly differed from other log events.

As of Erlang/OTP 21.0, the concept of SASL reports is removed, meaning that the default behaviour is as follows:

- Supervisor reports, crash reports, and progress reports are no longer connected to the SASL application.
- Supervisor reports and crash reports are issued as error level log events, and are logged through the default handler started by Kernel.
- Progress reports are issued as info level log events, and since the default primary log level is notice, these are not logged by default. To enable printing of progress reports, set the primary log level to info.
- The output format is the same for all log events.

If the old behaviour is preferred, the Kernel configuration parameter logger_sasl_compatible can be set to true. The SASL configuration parameters can then be used as before, and the SASL reports will only be printed if the SASL application is running, through a second log handler named sasl.

All SASL reports have a metadata field domain which is set to [otp,sasl]. This field can be used by filters to stop or allow the log events.
See section SASL User's Guide for more information about the old SASL error logging functionality.

Legacy Event Handlers

To use event handlers written for `error_logger`, just add your event handler with

```erlang
error_logger:add_report_handler/1,2.
```

This automatically starts the error logger event manager, and adds `error_logger` as a handler to Logger, with the following configuration:

```erlang
#{level => info,
  filter_default => log,
  filters => []}.
```

**Note:**
This handler ignores events that do not originate from the `error_logger` API, or from within OTP. This means that if your code uses the Logger API for logging, then your log events will be discarded by this handler.

The handler is not overload protected.

### 1.3.8 Error Handling

Logger does, to a certain extent, check its input data before forwarding a log event to filters and handlers. It does, however, not evaluate report callbacks, or check the validity of format strings and arguments. This means that all filters and handlers must be careful when formatting the data of a log event, making sure that it does not crash due to bad input data or faulty callbacks.

If a filter or handler still crashes, Logger will remove the filter or handler in question from the configuration, and print a short error message to the terminal. A debug event containing the crash reason and other details is also issued.

See section Log Message for more information about report callbacks and valid forms of log messages.

### 1.3.9 Example: Add a handler to log info events to file

When starting an Erlang node, the default behaviour is that all log events on level `notice` or more severe, are logged to the terminal via the default handler. To also log info events, you can either change the primary log level to `info`:

```erlang
1> logger:set_primary_config(level, info).
ok
```

or set the level for one or a few modules only:

```erlang
2> logger:set_module_level(mymodule, info).
ok
```

This allows info events to pass through to the default handler, and be printed to the terminal as well. If there are many info events, it can be useful to print these to a file instead.

First, set the log level of the default handler to `notice`, preventing it from printing info events to the terminal:

```erlang
3> logger:set_handler_config(default, level, notice).
ok
```

Then, add a new handler which prints to file. You can use the handler module `logger_std_h`, and configure it to log to file:
1.3 Logging

4> Config = #{config => #{file => "./info.log"}, level => info}.
#{config => #{file => "./info.log"},level => info}
5> logger:add_handler(myhandler, logger_std_h, Config).
ok

Since filter_default defaults to log, this handler now receives all log events. If you want info events only in the
file, you must add a filter to stop all non-info events. The built-in filter logger_filters:level/2 can do this:
6> logger:add_handler_filter(myhandler, stop_non_info,
{fun logger_filters:level/2, {stop, neq, info}}).
ok

See section Filters for more information about the filters and the filter_default configuration parameter.

1.3.10 Example: Implement a handler
Section Handler Callback Functions in the logger(3) manual page describes the callback functions that can be
implemented for a Logger handler.
A handler callback module must export:
•

log(Log, Config)

It can optionally also export some, or all, of the following:
•
•
•
•

adding_handler(Config)
removing_handler(Config)
changing_config(SetOrUpdate, OldConfig, NewConfig)
filter_config(Config)

When a handler is added, by for example a call to logger:add_handler(Id, HModule, Config),
Logger first calls HModule:adding_handler(Config). If this function returns {ok,Config1}, Logger
writes Config1 to the configuration database, and the logger:add_handler/3 call returns. After this, the
handler is installed and must be ready to receive log events as calls to HModule:log/2.
A
handler
can
be
removed
by
calling
logger:remove_handler(Id).
Logger
calls
HModule:removing_handler(Config), and removes the handler's configuration from the configuration
database.
When logger:set_handler_config/2,3 or logger:update_handler_config/2,3 is called,
Logger calls HModule:changing_config(SetOrUpdate, OldConfig, NewConfig). If this function
returns {ok,NewConfig1}, Logger writes NewConfig1 to the configuration database.
When logger:get_config/0 or logger:get_handler_config/0,1 is called, Logger calls
HModule:filter_config(Config). This function must return the handler configuration where internal data
is removed.
A simple handler that prints to the terminal can be implemented as follows:
-module(myhandler1).
-export([log/2]).
log(LogEvent, #{formatter := {FModule, FConfig}}) ->
io:put_chars(FModule:format(LogEvent, FConfig)).

Notice that the above handler does not have any overload protection, and all log events are printed directly from the
client process.

Ericsson AB. All Rights Reserved.: Kernel | 23


1.3 Logging

For information and examples of overload protection, please refer to section Protecting the Handler from Overload, and the implementation of `logger_std_h` and `logger_disk_log_h`.

The following is a simpler example of a handler which logs to a file through one single process:

```erlang
-module(myhandler2).
-export([adding_handler/1, removing_handler/1, log/2]).
-export([init/1, handle_call/3, handle_cast/2, terminate/2]).

adding_handler(Config) ->
    MyConfig = maps:get(config, Config, #{file => "myhandler2.log"}),
    {ok, Pid} = gen_server:start(?MODULE, MyConfig, []),
    {ok, Config#{config => MyConfig#{pid => Pid}}).

removing_handler(#{config := #{pid := Pid}}) ->
    gen_server:stop(Pid).

log(LogEvent,#{config := #{pid := Pid}} = Config) ->
    gen_server:cast(Pid, {log, LogEvent, Config}).

init(#{file := File}) ->
    {ok, Fd} = file:open(File, [append, {encoding, utf8}]),
    {ok, #{file => File, fd => Fd}}.

handle_call(_, _, State) ->
    {reply, {error, bad_request}, State}.

handle_cast({log, LogEvent, Config}, #{fd := Fd} = State) ->
    do_log(Fd, LogEvent, Config),
    {noreply, State}.

terminate(_Reason, #{fd := Fd}) ->
    _ = file:close(Fd),
    ok.

do_log(Fd, LogEvent, #{formatter := {FModule, FConfig}}) ->
    String = FModule:format(LogEvent, FConfig),
    io:put_chars(Fd, String).

1.3.11 Protecting the Handler from Overload

The default handlers, `logger_std_h` and `logger_disk_log_h`, feature an overload protection mechanism, which makes it possible for the handlers to survive, and stay responsive, during periods of high load (when huge numbers of incoming log requests must be handled). The mechanism works as follows:

Message Queue Length

The handler process keeps track of the length of its message queue and takes some form of action when the current length exceeds a configurable threshold. The purpose is to keep the handler in, or to as quickly as possible get the handler into, a state where it can keep up with the pace of incoming log events. The memory use of the handler must never grow larger and larger, since that will eventually cause the handler to crash. These three thresholds, with associated actions, exist:

- **sync_mode_qlen**
  
  As long as the length of the message queue is lower than this value, all log events are handled asynchronously. This means that the client process sending the log event, by calling a log function in the Logger API, does not wait for a response from the handler but continues executing immediately after the event is sent. It is not affected by the time it takes the handler to print the event to the log device. If the message queue grows larger than this value, the handler starts handling log events synchronously instead, meaning that the client process sending the event must wait for a response. When the handler reduces the message queue to a level below the `sync_mode_qlen`
threshold, asynchronous operation is resumed. The switch from asynchronous to synchronous mode can slow
down the logging tempo of one, or a few, busy senders, but cannot protect the handler sufficiently in a situation
of many busy concurrent senders.

Defaults to 10 messages.

**drop_mode_qlen**

When the message queue grows larger than this threshold, the handler switches to a mode in which it drops all
new events that senders want to log. Dropping an event in this mode means that the call to the log function never
results in a message being sent to the handler, but the function returns without taking any action. The handler
keeps logging the events that are already in its message queue, and when the length of the message queue is
reduced to a level below the threshold, synchronous or asynchronous mode is resumed. Notice that when the
handler activates or deactivates drop mode, information about it is printed in the log.

Defaults to 200 messages.

**flush_qlen**

If the length of the message queue grows larger than this threshold, a flush (delete) operation takes place. To
flush events, the handler discards the messages in the message queue by receiving them in a loop without logging.
Client processes waiting for a response from a synchronous log request receive a reply from the handler indicating
that the request is dropped. The handler process increases its priority during the flush loop to make sure that no
new events are received during the operation. Notice that after the flush operation is performed, the handler prints
information in the log about how many events have been deleted.

Defaults to 1000 messages.

For the overload protection algorithm to work properly, it is required that:

```
sync_mode_qlen =< drop_mode_qlen =< flush_qlen
```

and that:

```
drop_mode_qlen > 1
```

To disable certain modes, do the following:

- If `sync_mode_qlen` is set to 0, all log events are handled synchronously. That is, asynchronous logging is
disabled.
- If `sync_mode_qlen` is set to the same value as `drop_mode_qlen`, synchronous mode is disabled. That is,
the handler always runs in asynchronous mode, unless dropping or flushing is invoked.
- If `drop_mode_qlen` is set to the same value as `flush_qlen`, drop mode is disabled and can never occur.

During high load scenarios, the length of the handler message queue rarely grows in a linear and predictable way.
Instead, whenever the handler process is scheduled in, it can have an almost arbitrary number of messages waiting in
the message queue. It is for this reason that the overload protection mechanism is focused on acting quickly, and quite
drastically, such as immediately dropping or flushing messages, when a large queue length is detected.

The values of the previously listed thresholds can be specified by the user. This way, a handler can be configured
to, for example, not drop or flush messages unless the message queue length of the handler process grows extremely
large. Notice that large amounts of memory can be required for the node under such circumstances. Another example
of user configuration is when, for performance reasons, the client processes must never be blocked by synchronous
log requests. It is possible, perhaps, that dropping or flushing events is still acceptable, since it does not affect the
performance of the client processes sending the log events.

A configuration example:
### 1.3 Logging

```ruby
caller: add_handler(my_standard_h, logger_std_h,
  #{config => #{file => "./system_info.log",
    sync_mode_qlen => 100,
    drop_mode_qlen => 1000,
    flush_qlen => 2000}}).
```

**Controlling Bursts of Log Requests**

Large bursts of log events - many events received by the handler under a short period of time - can potentially cause problems, such as:

- Log files grow very large, very quickly.
- Circular logs wrap too quickly so that important data is overwritten.
- Write buffers grow large, which slows down file sync operations.

For this reason, both built-in handlers offer the possibility to specify the maximum number of events to be handled within a certain time frame. With this burst control feature enabled, the handler can avoid choking the log with massive amounts of printouts. The configuration parameters are:

- `burst_limit_enable`
  - Value `true` enables burst control and `false` disables it.
  - Defaults to `true`.

- `burst_limit_max_count`
  - This is the maximum number of events to handle within a `burst_limit_window_time` time frame. After the limit is reached, successive events are dropped until the end of the time frame.
  - Defaults to 500 events.

- `burst_limit_window_time`
  - See the previous description of `burst_limit_max_count`.
  - Defaults to 1000 milliseconds.

A configuration example:

```ruby
caller: add_handler(my_disk_log_h, logger_disk_log_h,
  #{config => #{file => "./my_disk_log",
    burst_limit_enable => true,
    burst_limit_max_count => 20,
    burst_limit_window_time => 500}}).
```

**Terminating an Overloaded Handler**

It is possible that a handler, even if it can successfully manage peaks of high load without crashing, can build up a large message queue, or use a large amount of memory. The overload protection mechanism includes an automatic termination and restart feature for the purpose of guaranteeing that a handler does not grow out of bounds. The feature is configured with the following parameters:

- `overload_kill_enable`
  - Value `true` enables the feature and `false` disables it.
  - Defaults to `false`.

- `overload_kill_qlen`
  - This is the maximum allowed queue length. If the message queue grows larger than this, the handler process is terminated.
  - Defaults to 20000 messages.
overload_kill_mem_size

This is the maximum memory size that the handler process is allowed to use. If the handler grows larger than
this, the process is terminated.
Defaults to 3000000 bytes.

overload_kill_restart_after

If the handler is terminated, it restarts automatically after a delay specified in milliseconds. The value infinity
prevents restarts.
Defaults to 5000 milliseconds.

If the handler process is terminated because of overload, it prints information about it in the log. It also prints
information about when a restart has taken place, and the handler is back in action.

Note:
The sizes of the log events affect the memory needs of the handler. For information about how to limit the size of
log events, see the logger_formatter(3) manual page.

1.3.12 Logger Proxy

The Logger proxy is an Erlang process which is part of the Kernel application's supervision tree. During startup, the
proxy process registers itself as the system_logger, meaning that log events produced by the emulator are sent
to this process.

When a log event is issued on a process which has its group leader on a remote node, Logger automatically forwards the
log event to the group leader's node. To achieve this, it first sends the log event as an Erlang message from the original
client process to the proxy on the local node, and the proxy in turn forwards the event to the proxy on the remote node.

When receiving a log event, either from the emulator or from a remote node, the proxy calls the Logger API to log
the event.

The proxy process is overload protected in the same way as described in section Protecting the Handler from Overload,
but with the following default values:

  #{sync_mode_qlen => 500, drop_mode_qlen => 1000, flush_qlen => 5000, burst_limit_enable => false, overload_kill_enable => false}

For log events from the emulator, synchronous message passing mode is not applicable, since all messages are passed
asynchronously by the emulator. Drop mode is achieved by setting the system_logger to undefined, forcing
the emulator to drop events until it is set back to the proxy pid again.

The proxy uses erlang:send_nosuspend/2 when sending log events to a remote node. If the message could
not be sent without suspending the sender, it is dropped. This is to avoid blocking the proxy process.

1.3.13 See Also

disk_log(3), erlang(3), error_logger(3), logger(3), logger_disk_log_h(3), logger_filters(3), logger_formatter(3), logger_std_h(3), sasl(6)
1.4 Logging Cookbook

Using and especially configuring Logger can be difficult at times as there are many different options that can be changed and often more than one way to achieve the same result. This User's Guide tries to help by giving many different examples of how you can use logger.

For more examples of practical use-cases of using Logger, Fred Hebert's blog post Erlang/OTP 21's new logger is a great starting point.

Note:

If you find that some common Logger usage is missing from this guide, please open a pull request on github with the suggested addition.

1.4.1 Get Logger information

Print the primary Logger configurations.

```
1> logger:i(primary).
Primary configuration:
  Level: notice
  Filter Default: log
  Filters: (none)
```

It is also possible to fetch the configuration using `logger:get_primary_config()`.

See also

- logger:i()
- Configuration in the Logging User's Guide
Print the configuration of all handlers.

```
2> logger:i(handlers).
Handler configuration:
   Id: default
     Module: logger_std_h
     Level: all
     Formatter:
       Module: logger_formatter
       Config:
         legacy_header: true
         single_line: false
     Filter Default: stop
     Filters:
       Id: remote_gl
         Fun: fun logger_filters:remote_gl/2
         Arg: stop
       Id: domain
         Fun: fun logger_filters:domain/2
         Arg: {log,super,[otp,sasl]}
       Id: no_domain
         Fun: fun logger_filters:domain/2
         Arg: {log,undefined,[]}
   Handler Config:
     burst_limit_enable: true
     burst_limit_max_count: 500
     burst_limit_window_time: 1000
     drop_mode_qlen: 200
     filesync_repeat_interval: no_repeat
     flush_qlen: 1000
     overload_kill_enable: false
     overload_kill_mem_size: 3000000
     overload_kill_qlen: 20000
     overload_kill_restart_after: 5000
     sync_mode_qlen: 10
     type: standard_io
```

You can also print the configuration of a specific handler using `logger:i(HandlerName)`, or fetch the configuration using `logger:get_handler_config()`, or `logger:get_handler_config(HandlerName)` for a specific handler.

See also
- `logger:i()`
- Configuration in the Logging User's Guide

### 1.4.2 Configure the Logger

**Where did my progress reports go?**

In OTP-21 the default primary log level is `notice`. The means that many log messages are by default not printed. This includes the progress reports of supervisors. In order to get progress reports you need to raise the primary log level to `info`.

```
$ erl -kernel logger_level info
=PROGRESS REPORT==== 4-Nov-2019::16:33:11.742069 ===
   application: kernel
   started_at: nonode@nohost
=PROGRESS REPORT==== 4-Nov-2019::16:33:11.746546 ===
   application: stdlib
   started_at: nonode@nohost
Eshell V10.5.3 (abort with ^G)
```
1.4.3 Configure Logger formatter

In order to fit better into your existing logging infrastructure Logger can format its logging messages any way you want to. Either you can use the built-in formatter, or you can build your own.

Single line configuration

Since single line logging is the default of the built-in formatter you only have to provide the empty map as the configuration. The example below uses the `sys.config` to change the formatter configuration.

```
$ cat sys.config
{{kernel,
  {{logger,
    {{handler, default, logger_std_h,
      #{ formatter => {logger_formatter, #{} } } } } } } } }.
```

```
$ erl -config sys
Eshell V10.5.1  (abort with ^G)
1> logger:error("Oh noes, an error").
1962-10-03T11:07:47.466763-04:00 error: Oh noes, an error
```

However, if you just want to change it for the current session you can also do that.

```
1> logger:set_handler_config(default, formatter, {logger_formatter, #{}}).
ok
2> logger:error("Oh noes, another error").
1962-10-04T15:34:02.648713-04:00 error: Oh noes, another error
```

See also

- `logger_formatter's Configuration`
- Formatters in the Logging User's Guide
- `logger:set_handler_config/3`

Add file and line number to log entries

You can change what is printed to the log by using the formatter template:

```
$ cat sys.config
{{kernel,
  {{logger,
    {{handler, default, logger_std_h,
      #{ formatter => {logger_formatter,
        \n" ] } } } } } } } } ]}.
```

```
$ erl -config sys
Eshell V10.5.1  (abort with ^G)
1> logger:error("Oh noes, more errors",#{ file => "shell.erl", line => 1 }).
1962-10-05T15:34:02.648713-04:00 shell.erl:1 error: Oh noes, more errors
```

Note that file and line have to be added in the metadata by the caller of `logger:log/3` as otherwise Logger will not know from where it was called. The file and line number are automatically added if you use the `?LOG_ERROR` macros in `kernel/include/logger.hrl`.

See also

- `logger_formatter's Configuration`
- `logger_formatter's Template`
- `Logger Macros`
- Metadata in the Logging User's Guide
1.4.4 Configuring handlers

Print logs to a file

Instead of printing the logs to stdout we print them to a rotating file log.

```erl
$ cat sys.config
{{kernel,
  [{logger,
    [{handler, default, logger_std_h,
      #{ config => #{ file => "log/erlang.log",
                        max_no_bytes => 4096,
                        max_no_files => 5},
        formatter => {logger_formatter, #{}}}},
     {}]]}}.
$ erl -config sys
Eshell V10.5.1 (abort with ^G)
1> logger:error("Oh noes, even more errors").
ok
2> erlang:halt().
$ cat log/erlang.log
2019-10-07T11:47:16.837958+02:00 error: Oh noes, even more errors
```

See also

- logger_std_h's Description
- Handlers in the Logging User's Guide

Debug only handler

Add a handler that prints debug log events to a file, while the default handler prints only up to notice level events to standard out.

```erl
$ cat sys.config
{{kernel,
  [{loggger_level, all},
   {logger,
    [{handler, default, logger_std_h,
      #{ level => notice }},
     {handler, debug, logger_std_h,
      #{ filters => [{debug,{fun logger_filters:level/2, {stop, neq, debug}}}],
        config => #{ file => "log/debug.log" } }]}}].
$ erl -config sys
Eshell V10.5.1 (abort with ^G)
1> logger:error("Oh noes, even more errors").
2> logger:debug("A debug event").
ok
3> erlang:halt().
$ cat log/debug.log
2019-10-07T11:41:03.680541+02:00 debug: A debug event
```

In the configuration above we first raise the primary log level to max in order for the debug log events to get to the handlers. Then we configure the default handler to only log notice and below events, the default log level for a handler is all. Then the debug handler is configured with a filter to stop any log message that is not a debug level message. It is also possible to do the same changes in an already running system using the logger module. Then you do like this:
1.4 Logging Cookbook

```
$ erl
1> logger:set_handler_config(default, level, notice).
ok
2> logger:add_handler(debug, logger_std_h, #{
    filters => [{debug, {fun logger_filters:level/2, {stop, neq, debug}}}],
    config => #{ file => "log/debug.log" } }).
ok
3> logger:set_primary_config(level, all).
ok
```

It is important that you do not raise the primary log level before adjusting the default handler's level as otherwise your standard out may be flooded by debug log messages.

See also
- logger_std_h's Description
- Filters in the Logging User's Guide

### 1.4.5 Logging

#### What to log and how

The simplest way to log something is by using the Logger macros and give a report to the macro. For example if you want to log an error:

```
?LOG_ERROR({ what => http_error, status => 418, src => ClientIP, dst => ServerIP }).
```

This will print the following in the default log:

```
dst: {8,8,4,4}
src: {8,8,8,8}
status: 418
what: http_error
```

or the below if you use a single line formatter:

```
2019-10-10T12:14:11.921843+02:00 error: dst: {8,8,4,4}, src: {8,8,8,8}, status: 418, what: http_error
```

See also
- Log Message in the Logging User's Guide

#### Report call-backs and printing of events

If you want to do structured logging, but still want to have some control of how the final log message is formatted you can give a report_cb as part of the metadata with your log event.

```
ReportCB = fun(#{ what := What, status := Status, src := Src, dst := Dst }) ->
    {ok, #hostent{ h_name = SrcName }} = inet:gethostbyaddr(Src),
    {ok, #hostent{ h_name = DstName }} = inet:gethostbyaddr(Dst),
    {"What: ~p~nStatus: ~p~nSrc: ~s (~s)~nDst: ~s (~s)~n",
        [What, Status, inet:ntoa(Src), SrcName, inet:ntoa(Dst), DstName]}
    end,
?LOG_ERROR(#{ what => http_error, status => 418, src => ClientIP, dst => ServerIP },
    #{ report_cb => ReportCB }).
```

This will print the following:
Note that the order that things are printed have changed, and also I added a reverse-dns lookup of the IP address. This will not print as nicely when using a single line formatter, however you can also use a report_cb fun with 2 arguments where the second argument is the formatting options.

See also
• Log Message in the Logging User's Guide
• Logger Report Callbacks

1.4.6 Filters

Filters are used to remove or change log events before they reach the handlers.

Process filters

If we only want debug messages from a specific process it is possible to do this with a filter like this:

```erlang
%% Initial setup to use a filter for the level filter instead of the primary level PrimaryLevel = maps:get(level, logger:get_primary_config()), ok = logger:add_primary_filter(primary_level,_), {fun logger_filters:level/2, {log, gteq, PrimaryLevel}}}, logger:set_primary_config(filter_default, stop), logger:set_primary_config(level, all),
%%% Test that things work as they should logger:notice("Notice should be logged"), logger:debug("Should not be logged"),
%%% Add the filter to allow PidToLog to send debug events PidToLog = self(), PidFilter = fun(LogEvent, _) when PidToLog =:= self() -> LogEvent; (_LogEvent, _) -> ignore end, ok = logger:add_primary_filter(pid, {PidFilter,[[]]}, logger:debug("Debug should be logged").
```

There is a bit of setup needed to allow filters to decide whether a specific process should be allowed to log. This is because the default primary log level is notice and it is enforced before the primary filters. So in order for the pid filter to be useful we have to raise the primary log level to all and then add a level filter that only lets certain messages at or greater than notice through. When the setup is done, it is simple to add a filter that allows a certain pid through.

Note that doing the primary log level filtering through a filter and not through the level is quite a lot more expensive, so make sure to test that your system can handle the extra load before you enable it on a production node.

See also
• Filters in the Logging User's Guide
• logger_filters:level/2
• logger:set_primary_config/2

Domains

Domains are used to specify which subsystem a certain log event originates from. The default handler will by default only log events with the domain [otp] or without a domain. If you would like to include SSL log events into the default handler log you could do this:
1.5 EEP-48: Documentation storage and format

This User's Guide describes the documentation storage format initially described in EEP-48. By standardizing how API documentation is stored, it will be possible to write tools that work across languages.

To fetch the EEP-48 documentation for a module you can use `code:get_doc/1`.

### 1.5.1 the "Docs" storage

To look for documentation for a module name example, a tool should:

**Example:**

1. Look for `example.beam` in the code path, parse the BEAM file and retrieve the Docs chunk. If the chunk is not available, it should look for "example.beam" in the code path and find the `doc/chunks/example.chunk` file in the application that defines the example module. If a .chunk file is not available, then documentation is not available.

2. The choice of using a chunk or the filesystem is completely up to the language or library. In both cases, the documentation can be added or removed at any moment by stripping the Docs chunk or by removing the `doc/chunks` directory.

3. For example, languages like Elixir and LFE attach the Docs chunk at compilation time, which can be controlled via a compiler flag. On the other hand, projects like OTP itself will likely generate the doc/chunks entries on a separate command, completely unrelated from code compilation.

### 1.5.2 the "Docs" format

In both storages, the documentation is written in the exactly same format: an Erlang term serialized to binary via `term_to_binary/1`. The term may be optionally compressed when serialized. It must follow the type specification below:

```erlang
{docs_v1,
 Anno :: erl_anno:anno(),
 BeamLanguage :: atom(),
 Format :: binary(),
 ModuleDoc :: #{DocLanguage := DocValue} | none | hidden,
 Metadata :: map(),
 Docs :: [{
   {Kind, Name, Arity},
   Anno :: erl_anno:anno(),
   Signature :: [binary()],
   Doc :: #{DocLanguage := DocValue} | none | hidden,
   Metadata :: map()
   }]} when DocLanguage :: binary(),
 DocValue :: binary() | term()
```

where in the root tuple we have:
Anno
annotation (line, column, file) of the definition itself (see \texttt{erl_anno(3)})

BeamLanguage
an atom representing the language, for example: erlang, elixir, lfe, alpaca, etc

Format
the mime type of the documentation, such as "text/markdown" or "application/erlang+html". For details of the format used by Erlang see the EEP-48 Chapter in Erl_Docgen's User's Guide.

ModuleDoc
a map with the documentation language as key, such as "en" or "pt_BR", and the documentation as a binary value. It may be the atom \texttt{none} in case there is no documentation or the atom \texttt{hidden} if documentation has been explicitly disabled for this entry.

Metadata
a map of atom keys with any term as value. This can be used to add annotations like the authors of a module, deprecated, or anything else a language or documentation tool may find relevant.

Docs
a list of documentation for other entities (such as functions and types) in the module.

For each entry in Docs, we have:

\{Kind, Name, Arity\}
the kind, name and arity identifying the function, callback, type, etc. The official entities are: function, type and callback. Other languages will add their own. For instance, Elixir and LFE may add macro.

Anno
annotation (line, column, file) of the module documentation or of the definition itself (see \texttt{erl_anno(3)}).

Signature
the signature of the entity. It is a list of binaries. Each entry represents a binary in the signature that can be joined with a whitespace or a newline. For example, [\texttt{binary_to_atom(Binary, Encoding)}], \texttt{when is_binary(Binary)} may be rendered as a single line or two lines. It exists exclusively for exhibition purposes.

Doc
a map with the documentation language as key, such as "en" or "pt_BR", and the documentation as a value. The documentation may either be a binary or any Erlang term, both described by Format. If it is an Erlang term, then the Format must be "application/erlang+SUFFIX", such as "application/erlang+html">> when the documentation is an Erlang representation of an HTML document. The Doc may also be atom \texttt{none} in case there is no documentation or the atom \texttt{hidden} if documentation has been explicitly disabled for this entry.

Metadata
a map of atom keys with any term as value.

This shared format is the heart of the EEP as it is what effectively allows cross-language collaboration.

The Metadata field exists to allow languages, tools and libraries to add custom information to each entry. This EEP documents the following metadata keys:

authors := [binary()]
a list of authors as binaries.
cross_references := [module() | {module(), \{Kind, Name, Arity\}}]
a list of modules or module entries that can be used as cross references when generating documentation.
deprecated := binary()
when present, it means the current entry is deprecated with a binary that represents the reason for deprecation and a recommendation to replace the deprecated code.
since := binary()
a binary representing the version such entry was added, such as "1.3.0">> or "20.0">>.
edit_url := binary()
a binary representing a URL to change to change the documentation itself.
Any key may be added to Metadata at any time. Keys that are frequently used by the community can be standardized in future versions.

1.5.3 See Also

erl_ann(3), shell_docs(3), EEP-48 Chapter in Erl_Docgen's User's Guide, code:get_doc/1
2 Reference Manual
The Kernel application has all the code necessary to run the Erlang runtime system: file servers, code servers, and so on. The Kernel application is the first application started. It is mandatory in the sense that the minimal system based on Erlang/OTP consists of Kernel and STDLIB. Kernel contains the following functional areas:

- Start, stop, supervision, configuration, and distribution of applications
- Code loading
- Logging
- Global name service
- Supervision of Erlang/OTP
- Communication with sockets
- Operating system interface

Logger Handlers

Two standard logger handlers are defined in the Kernel application. These are described in the Kernel User's Guide, and in the `logger_std_h(3)` and `logger_disk_log_h(3)` manual pages.

OS Signal Event Handler

Asynchronous OS signals may be subscribed to via the Kernel applications event manager (see OTP Design Principles and `gen_event(3)`) registered as `erl_signal_server`. A default signal handler is installed which handles the following signals:

- **sigusr1**
  - The default handler will halt Erlang and produce a crashdump with slogan "Received SIGUSR1". This is equivalent to calling `erlang:halt("Received SIGUSR1").`

- **sigquit**
  - The default handler will halt Erlang immediately. This is equivalent to calling `erlang:halt().`

- **sigterm**
  - The default handler will terminate Erlang normally. This is equivalent to calling `init:stop()`.

Events

Any event handler added to `erl_signal_server` must handle the following events.

- **sighup**
  - Hangup detected on controlling terminal or death of controlling process

- **sigquit**
  - Quit from keyboard

- **sigabrt**
  - Abort signal from abort

- **sigalrm**
  - Timer signal from alarm
sigterm
Termination signal
sigusr1
User-defined signal 1
sigusr2
User-defined signal 2
sigchld
Child process stopped or terminated
sigstop
Stop process
sigtstp
Stop typed at terminal

Setting OS signals are described in `os:os_signal/2`.

**Configuration**

The following configuration parameters are defined for the Kernel application. For more information about configuration parameters, see file `app(4)`.

distributed = [Distrib]

Specifies which applications that are distributed and on which nodes they are allowed to execute. In this parameter:
- `Distrib = {App,Nodes} | {App,Time,Nodes}`
- `App = atom()`
- `Time = integer()>0`
- `Nodes = [node() | {node(),...,node()}]`

The parameter is described in `application:load/2`.

dist_auto_connect = Value

Specifies when nodes are automatically connected. If this parameter is not specified, a node is always automatically connected, for example, when a message is to be sent to that node. `Value` is one of:
- `never`
  Connections are never automatically established, they must be explicitly connected. See `net_kernel(3)`.
- `once`
  Connections are established automatically, but only once per node. If a node goes down, it must thereafter be explicitly connected. See `net_kernel(3)`.

dist_listen = boolean()

Specifies whether this node should be listening for incoming distribution connections. Using this option implies that the node also is -hidden.

permissions = [Perm]

Specifies the default permission for applications when they are started. In this parameter:
- `Perm = {ApplName,Bool}`
- `ApplName = atom()`
Permissions are described in application:permit/2.

logger = [Config]

Specifies the configuration for Logger, except the primary log level, which is specified with logger_level, and the compatibility with SASL Error Logging, which is specified with logger_sasl_compatible.

The logger parameter is described in section Logging in the Kernel User’s Guide.

logger_level = Level

Specifies the primary log level for Logger. Log events with the same, or a more severe level, pass through the primary log level check. See section Logging in the Kernel User’s Guide for more information about Logger and log levels.

Level = emergency | alert | critical | error | warning | notice | info | debug | all | none

To change the primary log level at runtime, use logger:set_primary_config(level, Level).

Defaults to notice.

logger_sasl_compatible = true | false

Specifies if Logger behaves backwards compatible with the SASL error logging functionality from releases prior to Erlang/OTP 21.0.

If this parameter is set to true, the default Logger handler does not log any progress-, crash-, or supervisor reports. If the SASL application is then started, it adds a Logger handler named sasl, which logs these events according to values of the SASL configuration parameter sasl_error_logger and sasl_errlog_type.

See section Deprecated Error Logger Event Handlers and Configuration in the sasl(6) manual page for information about the SASL configuration parameters.

See section SASL Error Logging in the SASL User’s Guide, and section Backwards Compatibility with error_logger in the Kernel User’s Guide for information about the SASL error logging functionality, and how Logger can be backwards compatible with this.

Defaults to false.

Note:

If this parameter is set to true, sasl_errlog_type indicates that progress reports shall be logged, and the configured primary log level is notice or more severe, then SASL automatically sets the primary log level to info. That is, this setting can potentially overwrite the value of the Kernel configuration parameter logger_level. This is to allow progress reports, which have log level info, to be forwarded to the handlers.

global_groups = [GroupTuple]

Defines global groups, see global_group(3). In this parameter:

- GroupTuple = {GroupName, [Node]} | {GroupName, PublishType, [Node]}
- GroupName = atom()
- PublishType = normal | hidden
- Node = node()

inet_default_connect_options = [{Opt, Val}]

Specifies default options for connect sockets, see inet(3).
inet_default_listen_options = [{Opt, Val}]
  Specifies default options for listen (and accept) sockets, see inet(3).
{inet_dist_use_interface, ip_address()}
  If the host of an Erlang node has many network interfaces, this parameter specifies which one to listen on. For
  the type definition of ip_address(), see inet(3).
{inet_dist_listen_min, First} and {inet_dist_listen_max, Last}
  Defines the First..Last port range for the listener socket of a distributed Erlang node.
{inet_dist_listen_options, Opts}
  Defines a list of extra socket options to be used when opening the listening socket for a distributed Erlang node.
  See gen_tcp:listen/2.
{inet_dist_connect_options, Opts}
  Defines a list of extra socket options to be used when connecting to other distributed Erlang nodes. See
  gen_tcp:connect/4.
inet_parse_error_log = silent
  If set, no log events are issued when erroneous lines are found and skipped in the various Inet configuration files.
inetrc = Filename
  The name (string) of an Inet user configuration file. For details, see section Inet Configuration in the
  ERTS User's Guide.
net_setuptime = SetupTime
  SetupTime must be a positive integer or floating point number, and is interpreted as the maximum allowed
time for each network operation during connection setup to another Erlang node. The maximum allowed value
is 120. If higher values are specified, 120 is used. Default is 7 seconds if the variable is not specified, or if the
value is incorrect (for example, not a number).
  Notice that this value does not limit the total connection setup time, but rather each individual network operation
during the connection setup and handshake.
net_ticktime = TickTime
  Specifies the net_kernel tick time in seconds. This is the approximate time a connected node may be
unresponsive until it is considered down and thereby disconnected.
  Once every TickTime/4 seconds, each connected node is ticked if nothing has been sent to it during that last
TickTime/4 interval. A tick is a small package sent on the connection. A connected node is considered to be
down if no ticks or payload packages have been received during the last four TickTime/4 intervals. This
ensures that nodes that are not responding, for reasons such as hardware errors, are considered to be down.
  As the availability is only checked every TickTime/4 seconds, the actual time T a node have been unresponsive
when detected may vary between MinT and MaxT, where:

\[
\begin{align*}
MinT &= TickTime - TickTime / 4 \\
MaxT &= TickTime + TickTime / 4
\end{align*}
\]

  TickTime defaults to 60 seconds. Thus, 45 < T < 75 seconds.
  Notice that all communicating nodes are to have the same TickTime value specified, as it determines both the
frequency of outgoing ticks and the expected frequency of incoming ticks.
  Normally, a terminating node is detected immediately by the transport protocol (like TCP/IP).
shutdown_timeout = integer() | infinity

Specifies the time application_controller waits for an application to terminate during node shutdown. If the timer expires, application_controller brutally kills application_master of the hanging application. If this parameter is undefined, it defaults to infinity.

sync_nodes_mandatory = [NodeName]

Specifies which other nodes that must be alive for this node to start properly. If some node in the list does not start within the specified time, this node does not start either. If this parameter is undefined, it defaults to [].

sync_nodes_optional = [NodeName]

Specifies which other nodes that can be alive for this node to start properly. If some node in this list does not start within the specified time, this node starts anyway. If this parameter is undefined, it defaults to the empty list.

sync_nodes_timeout = integer() | infinity

Specifies the time (in milliseconds) that this node waits for the mandatory and optional nodes to start. If this parameter is undefined, no node synchronization is performed. This option ensures that global is synchronized.

start_distribution = true | false

Starts all distribution services, such as rpc, global, and net_kernel if the parameter is true. This parameter is to be set to false for systems who want to disable all distribution functionality.

Defaults to true.

start_dist_ac = true | false

Starts the dist_ac server if the parameter is true. This parameter is to be set to true for systems using distributed applications.

Defaults to false. If this parameter is undefined, the server is started if parameter distributed is set.

start_boot_server = true | false

Starts the boot_server if the parameter is true (see erl_boot_server(3)). This parameter is to be set to true in an embedded system using this service.

Defaults to false.

boot_server_slaves = [SlaveIP]

If configuration parameter start_boot_server is true, this parameter can be used to initialize boot_server with a list of slave IP addresses:

SlaveIP = string() | atom | {integer(),integer(),integer(),integer()},

where 0 <= integer() <= 255.

Examples of SlaveIP in atom, string, and tuple form:

'150.236.16.70', "150,236,16,70", {150,236,16,70}.

Defaults to [].

start_disk_log = true | false

Starts the disk_log_server if the parameter is true (see disk_log(3)). This parameter is to be set to true in an embedded system using this service.

Defaults to false.

start_pg = true | false

Starts the default pg scope server (see pg(3)) if the parameter is true. This parameter is to be set to true in an embedded system that uses this service.
Defaults to false.

\texttt{start\_pg2 = true | false}

Starts the pg2 server (see \texttt{pg2(3)}) if the parameter is true. This parameter is to be set to true in an embedded system that uses this service.

Defaults to false.

\texttt{start\_timer = true | false}

Starts the timer_server if the parameter is true (see \texttt{timer(3)}). This parameter is to be set to true in an embedded system using this service.

Defaults to false.

\texttt{shell\_history = enabled | disabled}

Specifies whether shell history should be logged to disk between usages of \texttt{erl}.

\texttt{shell\_history\_drop = [string()]}\n
Specific log lines that should not be persisted. For example \texttt{["q().", "init:stop()."]} will allow to ignore commands that shut the node down. Defaults to [].

\texttt{shell\_history\_file\_bytes = integer()}

how many bytes the shell should remember. By default, the value is set to 512kb, and the minimal value is 50kb.

\texttt{shell\_history\_path = string()}

Specifies where the shell history files will be stored. defaults to the user's cache directory as returned by \texttt{filename:basedir(user\_cache, "erlang-history")}.

\texttt{shutdown\_func = {Mod, Func}}

Where:
\begin{itemize}
  \item Mod = atom()
  \item Func = atom()
\end{itemize}

Sets a function that \texttt{application\_controller} calls when it starts to terminate. The function is called as \texttt{Mod:Func(Reason)}, where \texttt{Reason} is the terminate reason for \texttt{application\_controller}, and it must return as soon as possible for \texttt{application\_controller} to terminate properly.

\texttt{source\_search\_rules = [DirRule] | [SuffixRule]}

Where:
\begin{itemize}
  \item DirRule = {ObjDirSuffix, SrcDirSuffix}
  \item SuffixRule = {ObjSuffix, SrcSuffix, [DirRule]}
  \item ObjDirSuffix = string()
  \item SrcDirSuffix = string()
  \item ObjSuffix = string()
  \item SrcSuffix = string()
\end{itemize}

Specifies a list of rules for use by \texttt{filelib:find\_file/2} \texttt{filelib:find\_source/2} If this is set to some other value than the empty list, it replaces the default rules. Rules can be simple pairs of directory suffixes, such as \texttt{["ebin", "src"]}, which are used by \texttt{filelib:find\_file/2}, or triples specifying separate directory suffix rules depending on file name extensions, for example \texttt{[[".beam", ".erl", \{"ebin", "src"\}]}], which are used by \texttt{filelib:find\_source/2}. Both kinds of rules can be mixed in the list.

The interpretation of \texttt{ObjDirSuffix} and \texttt{SrcDirSuffix} is as follows: if the end of the directory name where an object is located matches \texttt{ObjDirSuffix}, then the name created by replacing \texttt{ObjDirSuffix} with
SrcDirSuffix is expanded by calling filelib:wildcard/1, and the first regular file found among the matches is the source file.

Deprecated Configuration Parameters

In Erlang/OTP 21.0, a new API for logging was added. The old error_logger event manager, and event handlers running on this manager, still work, but they are no longer used by default.

The following application configuration parameters can still be set, but they are only used if the corresponding configuration parameters for Logger are not set.

error_logger

Replaced by setting the type, and possibly file and modes parameters of the default logger_std_h handler. Example:

```
erl -kernel logger '[[handler,default,logger_std_h,#{config=>#{file=>"/tmp/erlang.log"}}]]'
```

error_logger_format_depth

Replaced by setting the depth parameter of the default handlers formatter. Example:

```
erl -kernel logger '[[handler,default,logger_std_h,#{formatter=>{logger_formatter,#{legacy_header=>true,template=>
}}}]'
```

See Backwards compatibility with error_logger for more information.

See Also

The application resource file specifies the resources an application uses, and how the application is started. There must always be one application resource file called Application.app for each application Application in the system.

The file is read by the application controller when an application is loaded/started. It is also used by the functions in systools, for example when generating start scripts.

File Syntax

The application resource file is to be called Application.app, where Application is the application name. The file is to be located in directory ebin for the application.

The file must contain a single Erlang term, which is called an application specification:

```erlang
{application, Application, [
  {description, Description},
  {id, Id},
  {vsn, Vsn},
  {modules, Modules},
  {maxP, MaxP},
  {maxT, MaxT},
  {registered, Names},
  {included_applications, Apps},
  {applications, Apps},
  {env, Env},
  {mod, Start},
  {start_phases, Phases},
  {runtime_dependencies, RTDeps}
]}.  
```

<table>
<thead>
<tr>
<th>Value</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>atom()</td>
</tr>
<tr>
<td>Description</td>
<td>string()</td>
</tr>
<tr>
<td>Id</td>
<td>string()</td>
</tr>
<tr>
<td>Vsn</td>
<td>string()</td>
</tr>
<tr>
<td>Modules</td>
<td>[Module]</td>
</tr>
<tr>
<td>MaxP</td>
<td>int()</td>
</tr>
<tr>
<td>MaxT</td>
<td>int()</td>
</tr>
<tr>
<td>Names</td>
<td>[Name]</td>
</tr>
<tr>
<td>Apps</td>
<td>[App]</td>
</tr>
<tr>
<td>Env</td>
<td>[{Par,Val}]</td>
</tr>
<tr>
<td>Start</td>
<td>{Module,StartArgs}</td>
</tr>
<tr>
<td>Phases</td>
<td>[{Phase,PhaseArgs}]</td>
</tr>
<tr>
<td>RTDeps</td>
<td>[ApplicationVersion]</td>
</tr>
</tbody>
</table>

Module = Name = App = Par = Phase = atom()
Val = StartArgs = PhaseArgs = term()
ApplicationVersion = string()

For the application controller, all keys are optional. The respective default values are used for any omitted keys.

The functions in systools require more information. If they are used, the following keys are mandatory:

- description
The other keys are ignored by systools.

description
   A one-line description of the application.

id
   Product identification, or similar.

vsn
   Version of the application.

modules
   All modules introduced by this application. systools uses this list when generating start scripts and tar files. A module can only be defined in one application.

maxP
   Deprecated - is ignored
   Maximum number of processes allowed in the application.

maxT
   Maximum time, in milliseconds, that the application is allowed to run. After the specified time, the application terminates automatically.

registered
   All names of registered processes started in this application. systools uses this list to detect name clashes between different applications.

included_applications
   All applications included by this application. When this application is started, all included applications are loaded automatically, but not started, by the application controller. It is assumed that the top-most supervisor of the included application is started by a supervisor of this application.

applications
   All applications that must be started before this application is allowed to be started. systools uses this list to generate correct start scripts. Defaults to the empty list, but notice that all applications have dependencies to (at least) Kernel and STDLIB.

env
   Configuration parameters used by the application. The value of a configuration parameter is retrieved by calling application:get_env/1,2. The values in the application resource file can be overridden by values in a configuration file (see config(4)) or by command-line flags (see erts:erl(1)).

mod
   Specifies the application callback module and a start argument, see application(3).
   Key mod is necessary for an application implemented as a supervision tree, otherwise the application controller does not know how to start it. mod can be omitted for applications without processes, typically code libraries, for example, STDLIB.
**start_phases**

A list of start phases and corresponding start arguments for the application. If this key is present, the application master, in addition to the usual call to `Module:start/2`, also calls `Module:start_phase(Phase,Type,PhaseArgs)` for each start phase defined by key `start_phases`. Only after this extended start procedure, `application:start(Application)` returns.

Start phases can be used to synchronize startup of an application and its included applications. In this case, key `mod` must be specified as follows:

```
{mod, {application_starter,[Module,StartArgs]}}
```

The application master then calls `Module:start/2` for the primary application, followed by calls to `Module:start_phase/3` for each start phase (as defined for the primary application), both for the primary application and for each of its included applications, for which the start phase is defined.

This implies that for an included application, the set of start phases must be a subset of the set of phases defined for the primary application. For more information, see OTP Design Principles.

**runtime_dependencies**

A list of application versions that the application depends on. An example of such an application version is "kernel-3.0". Application versions specified as runtime dependencies are minimum requirements. That is, a larger application version than the one specified in the dependency satisfies the requirement. For information about how to compare application versions, see section Versions in the System Principles User's Guide.

Notice that the application version specifies a source code version. One more, indirect, requirement is that the installed binary application of the specified version is built so that it is compatible with the rest of the system.

Some dependencies can only be required in specific runtime scenarios. When such optional dependencies exist, these are specified and documented in the corresponding "App" documentation of the specific application.

**Warning:**

The `runtime_dependencies` key was introduced in OTP 17.0. The type of its value might be subject to changes during the OTP 17 release.

**Warning:**

All runtime dependencies specified in OTP applications during the OTP 17 release may not be completely correct. This is actively being worked on. Declared runtime dependencies in OTP applications are expected to be correct in OTP 18.

**See Also**

`application(3), systools(3)`
In OTP, application denotes a component implementing some specific functionality, that can be started and stopped as a unit, and that can be reused in other systems. This module interacts with application controller, a process started at every Erlang runtime system. This module contains functions for controlling applications (for example, starting and stopping applications), and functions to access information about applications (for example, configuration parameters).

An application is defined by an application specification. The specification is normally located in an application resource file named Application.app, where Application is the application name. For details about the application specification, see app (4).

This module can also be viewed as a behaviour for an application implemented according to the OTP design principles as a supervision tree. The definition of how to start and stop the tree is to be located in an application callback module, exporting a predefined set of functions.

For details about applications and behaviours, see OTP Design Principles.

Data Types

\[
\begin{align*}
\text{start_type}() &= \text{normal} \mid \{\text{takeover}, \text{Node} :: \text{node}()\} \mid \{\text{failover}, \text{Node} :: \text{node}()\} \\
\text{restart_type}() &= \text{permanent} \mid \text{transient} \mid \text{temporary} \\
\text{tuple_of}(T) &= \text{A tuple where the elements are of type } T.
\end{align*}
\]

Exports

\[
\begin{align*}
\text{ensure_all_started}(\text{Application}) &\rightarrow \{\text{ok}, \text{Started}\} \mid \{\text{error}, \text{Reason}\} \\
\text{ensure_all_started}(\text{Application}, \text{Type}) &\rightarrow \{\text{ok}, \text{Started}\} \mid \{\text{error}, \text{Reason}\}
\end{align*}
\]

Types:

\[
\begin{align*}
\text{Application} &= \text{atom()} \\
\text{Type} &= \text{restart_type()} \\
\text{Started} &= [\text{atom()}] \\
\text{Reason} &= \text{term()}
\end{align*}
\]

Equivalent to calling start/1,2 repeatedly on all dependencies that are not yet started for an application.

Returns \{ok, AppNames\} for a successful start or for an already started application (which is, however, omitted from the AppNames list).

The function reports \{error, \{AppName, Reason\}\} for errors, where Reason is any possible reason returned by start/1,2 when starting a specific dependency.

If an error occurs, the applications started by the function are stopped to bring the set of running applications back to its initial state.
ensure_started(Application) -> ok | {error, Reason}
ensure_started(Application, Type) -> ok | {error, Reason}

Types:
  Application = atom()
  Type = restart_type()
  Reason = term()

Equivalent to start/1,2 except it returns ok for already started applications.

get_all_env() -> Env
get_all_env(Application) -> Env

Types:
  Application = atom()
  Env = [{Par :: atom(), Val :: term()}]

Returns the configuration parameters and their values for Application. If the argument is omitted, it defaults to the application of the calling process.

If the specified application is not loaded, or if the process executing the call does not belong to any application, the function returns [].

get_all_key() -> [] | {ok, Keys}
get_all_key(Application) -> undefined | Keys

Types:
  Application = atom()
  Keys = {ok, [{Key :: atom(), Val :: term()}, ...]}

Returns the application specification keys and their values for Application. If the argument is omitted, it defaults to the application of the calling process.

If the specified application is not loaded, the function returns undefined. If the process executing the call does not belong to any application, the function returns [].

get_application() -> undefined | {ok, Application}
get_application(PidOrModule) -> undefined | {ok, Application}

Types:
  PidOrModule = (Pid :: pid()) | (Module :: module())
  Application = atom()

Returns the name of the application to which the process Pid or the module Module belongs. Providing no argument is the same as calling get_application(self()).

If the specified process does not belong to any application, or if the specified process or module does not exist, the function returns undefined.

get_env(Par) -> undefined | {ok, Val}
get_env(Application, Par) -> undefined | {ok, Val}

Types:
Application = Par = atom()
Val = term()

Returns the value of configuration parameter Par for Application. If the application argument is omitted, it defaults to the application of the calling process.

Returns undefined if any of the following applies:
• The specified application is not loaded.
• The configuration parameter does not exist.
• The process executing the call does not belong to any application.

get_env(Application, Par, Def) -> Val
Types:
    Application = Par = atom()
    Def = Val = term()

Works like get_env/2 but returns value Def when configuration parameter Par does not exist.

get_key(Key) -> undefined | {ok, Val}
get_key(Application, Key) -> undefined | {ok, Val}
Types:
    Application = Key = atom()
    Val = term()

Returns the value of the application specification key Key for Application. If the application argument is omitted, it defaults to the application of the calling process.

Returns undefined if any of the following applies:
• The specified application is not loaded.
• The specification key does not exist.
• The process executing the call does not belong to any application.

load(AppDescr) -> ok | {error, Reason}
load(AppDescr, Distributed) -> ok | {error, Reason}
Types:
    AppDescr = Application | (AppSpec :: application_spec())
    Application = atom()
    Distributed =
        {Application, Nodes} | {Application, Time, Nodes} | default
        Nodes = [node() | tuple_of(node())]
        Time = integer() >= 1
        Reason = term()
    application_spec() =
        {application,
            Application :: atom(),
            AppSpecKeys :: [application_opt()]}
    application_opt() =
        {description, Description :: string() |
Loads the application specification for an application into the application controller. It also loads the application specifications for any included applications. Notice that the function does not load the Erlang object code.

The application can be specified by its name `Application`. In this case, the application controller searches the code path for the application resource file `Application.app` and loads the specification it contains.

The application specification can also be specified directly as a tuple `AppSpec`, having the format and contents as described in `app(4)`.

If `Distributed == {Application, [Time,] Nodes}`, the application becomes distributed. The argument overrides the value for the application in the Kernel configuration parameter `distributed`. `Application` must be the application name (same as in the first argument). If a node crashes and `Time` is specified, the application controller waits for `Time` milliseconds before attempting to restart the application on another node. If `Time` is not specified, it defaults to 0 and the application is restarted immediately.

`Nodes` is a list of node names where the application can run, in priority from left to right. Node names can be grouped using tuples to indicate that they have the same priority.

**Example:**

```
Nodes = [cp1@cave, {cp2@cave, cp3@cave}]
```

This means that the application is preferably to be started at `cp1@cave`. If `cp1@cave` is down, the application is to be started at `cp2@cave` or `cp3@cave`.

If `Distributed == default`, the value for the application in the Kernel configuration parameter `distributed` is used.

```erlang
loaded_applications() -> [{Application, Description, Vsn}]
```

**Types:**

- `Application = atom()`
- `Description = Vsn = string()`

Returns a list with information about the applications, and included applications, which are loaded using `load/1,2`. `Application` is the application name. `Description` and `Vsn` are the values of their description and `vsn` application specification keys, respectively.

```erlang
set_env(Config) -> ok
set_env(Config, Opts) -> ok
```

**Types:**
Config = [{Application, Env}]
  Application = atom()
  Env = [{Par :: atom(), Val :: term()}]
  Opts = [{timeout, timeout()} | {persistent, boolean()}]

Sets the configuration Config for multiple applications. It is equivalent to calling set_env/4 on each application individually, except it is more efficient. The given Config is validated before the configuration is set.

set_env/2 uses the standard gen_server time-out value (5000 ms). Option timeout can be specified if another time-out value is useful, for example, in situations where the application controller is heavily loaded.

Option persistent can be set to true to guarantee that parameters set with set_env/2 are not overridden by those defined in the application resource file on load. This means that persistent values will stick after the application is loaded and also on application reload.

If an application is given more than once or if an application has the same key given more than once, the behaviour is undefined and a warning message will be logged. In future releases, an error will be raised.

set_env/1 is equivalent to set_env(Config, []).  

Warning:

Use this function only if you know what you are doing, that is, on your own applications. It is very application-dependent and configuration parameter-dependent when and how often the value is read by the application. Careless use of this function can put the application in a weird, inconsistent, and malfunctioning state.

permit(Application, Permission) -> ok | {error, Reason}

Types:
  Application = atom()
  Permission = boolean()
  Reason = term()

Changes the permission for Application to run at the current node. The application must be loaded using load/1,2 for the function to have effect.

If the permission of a loaded, but not started, application is set to false, start returns ok but the application is not started until the permission is set to true.

If the permission of a running application is set to false, the application is stopped. If the permission later is set to true, it is restarted.

If the application is distributed, setting the permission to false means that the application will be started at, or moved to, another node according to how its distribution is configured (see load/2).

The function does not return until the application is started, stopped, or successfully moved to another node. However, in some cases where permission is set to true, the function returns ok even though the application is not started. This is true when an application cannot start because of dependencies to other applications that are not yet started. When they are started, Application is started as well.

By default, all applications are loaded with permission true on all nodes. The permission can be configured using the Kernel configuration parameter permissions.

set_env(Application, Par, Val) -> ok

set_env(Application, Par, Val, Opts) -> ok

Types:
Application = Par = atom()
Val = term()
Opts = [{timeout, timeout()} | {persistent, boolean()}]

Sets the value of configuration parameter Par for Application.

set_env/4 uses the standard gen_server time-out value (5000 ms). Option timeout can be specified if another
time-out value is useful, for example, in situations where the application controller is heavily loaded.

If set_env/4 is called before the application is loaded, the application environment values specified in file
Application.app override the ones previously set. This is also true for application reloads.

Option persistent can be set to true to guarantee that parameters set with set_env/4 are not overridden by
those defined in the application resource file on load. This means that persistent values will stick after the application
is loaded and also on application reload.

Warning:

Use this function only if you know what you are doing, that is, on your own applications. It is very application-
dependent and configuration parameter-dependent when and how often the value is read by the application. Careless
use of this function can put the application in a weird, inconsistent, and malfunctioning state.

start(Application) -> ok | {error, Reason}
start(Application, Type) -> ok | {error, Reason}

Types:
Application = atom()
Type = restart_type()
Reason = term()

Starts Application. If it is not loaded, the application controller first loads it using load/1. It ensures that
any included applications are loaded, but does not start them. That is assumed to be taken care of in the code for
Application.

The application controller checks the value of the application specification key applications, to ensure that all
applications needed to be started before this application are running. Otherwise, {error, {not_started, App}}
is returned, where App is the name of the missing application.

The application controller then creates an application master for the application. The application master becomes
the group leader of all the processes in the application. I/O is forwarded to the previous group leader, though, this is
just a way to identify processes that belong to the application. Used for example to find itself from any process, or,
reciprocally, to kill them all when it terminates.

The application master starts the application by calling the application callback function Module:start/2 as
defined by the application specification key mod.

Argument Type specifies the type of the application. If omitted, it defaults to temporary.

• If a permanent application terminates, all other applications and the entire Erlang node are also terminated.
• • If a transient application terminates with Reason == normal, this is reported but no other applications
are terminated.
• • If a transient application terminates abnormally, all other applications and the entire Erlang node are also
terminated.
• If a temporary application terminates, this is reported but no other applications are terminated.
Notice that an application can always be stopped explicitly by calling `stop/1`. Regardless of the type of the application, no other applications are affected.

Notice also that the transient type is of little practical use, because when a supervision tree terminates, the reason is set to `shutdown`, not `normal`.

```erlang
start_type() -> StartType | undefined | local
```

Types:
- `StartType = start_type()`

This function is intended to be called by a process belonging to an application, when the application is started, to determine the start type, which is `StartType` or `local`.

For a description of `StartType`, see `Module:start/2`.

`local` is returned if only parts of the application are restarted (by a supervisor), or if the function is called outside a startup.

If the process executing the call does not belong to any application, the function returns `undefined`.

```erlang
stop(Application) -> ok | {error, Reason}
```

Types:
- `Application = atom()`
- `Reason = term()`

Stops `Application`. The application master calls `Module:prep_stop/1`, if such a function is defined, and then tells the top supervisor of the application to shut down (see `supervisor/3`). This means that the entire supervision tree, including included applications, is terminated in reversed start order. After the shutdown, the application master calls `Module:stop/1`. `Module` is the callback module as defined by the application specification key `mod`.

Last, the application master terminates. Notice that all processes with the application master as group leader, that is, processes spawned from a process belonging to the application, are also terminated.

When stopped, the application is still loaded.

To stop a distributed application, `stop/1` must be called on all nodes where it can execute (that is, on all nodes where it has been started). The call to `stop/1` on the node where the application currently executes stops its execution. The application is not moved between nodes, as `stop/1` is called on the node where the application currently executes before `stop/1` is called on the other nodes.

```erlang
takeover(Application, Type) -> ok | {error, Reason}
```

Types:
- `Application = atom()`
- `Type = restart_type()`
- `Reason = term()`

Takes over the distributed application `Application`, which executes at another node `Node`. At the current node, the application is restarted by calling `Module:start([{takeover,Node},StartArgs])`. `Module` and `StartArgs` are retrieved from the loaded application specification. The application at the other node is not stopped until the startup is completed, that is, when `Module:start/2` and any calls to `Module:start_phase/3` have returned.

Thus, two instances of the application run simultaneously during the takeover, so that data can be transferred from the old to the new instance. If this is not an acceptable behavior, parts of the old instance can be shut down when the new instance is started. However, the application cannot be stopped entirely, at least the top supervisor must remain alive.
For a description of Type, see start/1,2.

unload(Application) -> ok | {error, Reason}
Types:
   Application = atom()
   Reason = term()
Unloads the application specification for Application from the application controller. It also unloads the application specifications for any included applications. Notice that the function does not purge the Erlang object code.

unset_env(Application, Par) -> ok
unset_env(Application, Par, Opts) -> ok
Types:
   Application = Par = atom()
   Opts = [{timeout, timeout()} | {persistent, boolean()}]
Removes the configuration parameter Par and its value for Application.
unset_env/2 uses the standard gen_server time-out value (5000 ms). Option timeout can be specified if another time-out value is useful, for example, in situations where the application controller is heavily loaded.
unset_env/3 also allows the persistent option to be passed (see set_env/4).

### Warning:
Use this function only if you know what you are doing, that is, on your own applications. It is very application-dependent and configuration parameter-dependent when and how often the value is read by the application. Careless use of this function can put the application in a weird, inconsistent, and malfunctioning state.

which_applications() -> [{Application, Description, Vsn}]
which_applications(Timeout) -> [{Application, Description, Vsn}]
Types:
   Timeout = timeout()
   Application = atom()
   Description = Vsn = string()
Returns a list with information about the applications that are currently running. Application is the application name. Description and Vsn are the values of their description and vsn application specification keys, respectively.
which_applications/0 uses the standard gen_server time-out value (5000 ms). A Timeout argument can be specified if another time-out value is useful, for example, in situations where the application controller is heavily loaded.
The following functions are to be exported from an application callback module.

**Exports**

Module:start(StartType, StartArgs) -> {ok, Pid} | {ok, Pid, State} | {error, Reason}
Types:
This function is called whenever an application is started using \texttt{start/1,2}, and is to start the processes of the application. If the application is structured according to the OTP design principles as a supervision tree, this means starting the top supervisor of the tree.

\texttt{StartType} defines the type of start:

\begin{itemize}
  \item \texttt{normal} if it is a normal startup.
  \item \texttt{normal} also if the application is distributed and started at the current node because of a failover from another node, and the application specification key \texttt{start\_phases} == \texttt{undefined}.
  \item \{\texttt{takeover}, \texttt{Node}\} if the application is distributed and started at the current node because of a takeover from \texttt{Node}, either because \texttt{takeover/2} has been called or because the current node has higher priority than \texttt{Node}.
  \item \{\texttt{failover}, \texttt{Node}\} if the application is distributed and started at the current node because of a failover from \texttt{Node}, and the application specification key \texttt{start\_phases} /= \texttt{undefined}.
\end{itemize}

\texttt{StartArgs} is the \texttt{StartArgs} argument defined by the application specification key \texttt{mod}.

The function is to return \{\texttt{ok,Pid}\} or \{\texttt{ok,Pid,State}\}, where \texttt{Pid} is the pid of the top supervisor and \texttt{State} is any term. If omitted, \texttt{State} defaults to \texttt{[]}. If the application is stopped later, \texttt{State} is passed to \texttt{Module:prep\_stop/1}.

\texttt{Module:start\_phase(Phase, StartType, PhaseArgs) -> ok | \{error, Reason\}}

\texttt{Types:}
\begin{itemize}
  \item \texttt{Phase = atom()}
  \item \texttt{StartType = start\_type()}
  \item \texttt{PhaseArgs = term()}
  \item \texttt{Pid = pid()}
  \item \texttt{State = state()}
\end{itemize}

Starts an application with included applications, when synchronization is needed between processes in the different applications during startup.

The start phases are defined by the application specification key \texttt{start\_phases} == \{\{\texttt{Phase}, \texttt{PhaseArgs}\}\}. For included applications, the set of phases must be a subset of the set of phases defined for the including application.

The function is called for each start phase (as defined for the primary application) for the primary application and all included applications, for which the start phase is defined.

For a description of \texttt{StartType}, see \texttt{Module:start/2}.

\texttt{Module:prep\_stop(State) -> NewState}

\texttt{Types:}
\begin{itemize}
  \item \texttt{State = NewState = term()}
\end{itemize}

This function is called when an application is about to be stopped, before shutting down the processes of the application. \texttt{State} is the state returned from \texttt{Module:start\_2}, or \texttt{[]} if no state was returned. \texttt{NewState} is any term and is passed to \texttt{Module:stop/1}.

The function is optional. If it is not defined, the processes are terminated and then \texttt{Module:stop(State)} is called.
Module:stop(State)

Types:

    State = term()

This function is called whenever an application has stopped. It is intended to be the opposite of Module:start/2 and is to do any necessary cleaning up. The return value is ignored.

State is the return value of Module:prep_stop/1, if such a function exists. Otherwise State is taken from the return value of Module:start/2.

Module:config_change(Changed, New, Removed) -> ok

Types:

    Changed = [{Par, Val}]
    New = [{Par, Val}]
    Removed = [Par]
    Par = atom()
    Val = term()

This function is called by an application after a code replacement, if the configuration parameters have changed.

Changed is a list of parameter-value tuples including all configuration parameters with changed values.

New is a list of parameter-value tuples including all added configuration parameters.

Removed is a list of all removed parameters.

See Also

OTP Design Principles, kernel(6), app(4)
auth

This module is deprecated. For a description of the Magic Cookie system, refer to Distributed Erlang in the Erlang Reference Manual.

Data Types

cookie() = atom()

Exports

cookie() -> Cookie
Types:
   Cookie = cookie()
Use erlang:get_cookie() in ERTS instead.

cookie(TheCookie) -> true
Types:
   TheCookie = Cookie | [Cookie]
   The cookie can also be specified as a list with a single atom element.
   Cookie = cookie()
Use erlang:set_cookie(node(), Cookie) in ERTS instead.

is_auth(Node) -> yes | no
Types:
   Node = node()
Returns yes if communication with Node is authorized. Notice that a connection to Node is established in this case. Returns no if Node does not exist or communication is not authorized (it has another cookie than auth thinks it has). Use net_adm:ping(Node) instead.

node_cookie([Node, Cookie]) -> yes | no
Types:
   Node = node()
   Cookie = cookie()
Equivalent to node_cookie(Node, Cookie).

node_cookie(Node, Cookie) -> yes | no
Types:
   Node = node()
   Cookie = cookie()
Sets the magic cookie of Node to Cookie and verifies the status of the authorization. Equivalent to calling erlang:set_cookie(Node, Cookie), followed by auth:is_auth(Node).
**code**

Erlang module

This module contains the interface to the Erlang code server, which deals with the loading of compiled code into a running Erlang runtime system.

The runtime system can be started in **interactive** or **embedded** mode. Which one is decided by the command-line flag `-mode`:

```erlang
% erl -mode interactive
```

The modes are as follows:

- In interactive mode, which is default, only some code is loaded during system startup, basically the modules needed by the runtime system. Other code is dynamically loaded when first referenced. When a call to a function in a certain module is made, and the module is not loaded, the code server searches for and tries to load the module.
- In embedded mode, modules are not auto loaded. Trying to use a module that has not been loaded results in an error. This mode is recommended when the boot script loads all modules, as it is typically done in OTP releases. (Code can still be loaded later by explicitly ordering the code server to do so).

To prevent accidentally reloading of modules affecting the Erlang runtime system, directories `kernel`, `stdlib`, and `compiler` are considered **sticky**. This means that the system issues a warning and rejects the request if a user tries to reload a module residing in any of them. The feature can be disabled by using command-line flag `-nostick`.

**Code Path**

In interactive mode, the code server maintains a search path, usually called the **code path**, consisting of a list of directories, which it searches sequentially when trying to load a module.

Initially, the code path consists of the current working directory and all Erlang object code directories under library directory `$OTP/ROOT/lib`, where `$OTP/ROOT` is the installation directory of Erlang/OTP, `code:root_dir()`. Directories can be named `Name[-Vsn]` and the code server, by default, chooses the directory with the highest version number among those having the same `Name`. Suffix `-Vsn` is optional. If an `ebin` directory exists under `Name[-Vsn]`, this directory is added to the code path.

Environment variable `ERL_LIBS` (defined in the operating system) can be used to define more library directories to be handled in the same way as the standard OTP library directory described above, except that directories without an `ebin` directory are ignored.

All application directories found in the additional directories appears before the standard OTP applications, except for the Kernel and STDLIB applications, which are placed before any additional applications. In other words, modules found in any of the additional library directories override modules with the same name in OTP, except for modules in Kernel and STDLIB.

Environment variable `ERL_LIBS` (if defined) is to contain a colon-separated (for Unix-like systems) or semicolon-separated (for Windows) list of additional libraries.

**Example:**

On a Unix-like system, `ERL_LIBS` can be set to the following

```
/usr/local/jungerl:/home/some_user/my_erlang_lib
```

On Windows, use semi-colon as separator.
Loading of Code From Archive Files

**Warning:**
The support for loading code from archive files is experimental. The purpose of releasing it before it is ready is to obtain early feedback. The file format, semantics, interfaces, and so on, can be changed in a future release. The function `lib_dir/2` and flag `-code_path_choice` are also experimental.

The Erlang archives are ZIP files with extension `.ez`. Erlang archives can also be enclosed in escript files whose file extension is arbitrary.

Erlang archive files can contain entire Erlang applications or parts of applications. The structure in an archive file is the same as the directory structure for an application. If you, for example, create an archive of `mnesia-4.4.7`, the archive file must be named `mnesia-4.4.7.ez` and it must contain a top directory named `mnesia-4.4.7`. If the version part of the name is omitted, it must also be omitted in the archive. That is, a `mnesia.ez` archive must contain a `mnesia` top directory.

An archive file for an application can, for example, be created like this:

```erlang
zip:create("mnesia-4.4.7.ez", [
    ["mnesia-4.4.7"],
    [{cwd, code:lib_dir()},
     {compress, all},
     {uncompress, [".beam",".app"]}],
])
```

Any file in the archive can be compressed, but to speed up the access of frequently read files, it can be a good idea to store `.beam` and `.app` files uncompressed in the archive.

Normally the top directory of an application is located in library directory `$OTPROOT/lib` or in a directory referred to by environment variable `ERL_LIBS`. At startup, when the initial code path is computed, the code server also looks for archive files in these directories and possibly adds `ebin` directories in archives to the code path. The code path then contains paths to directories that look like `$OTPROOT/lib/mnesia.ez/mnesia/ebin` or `$OTPROOT/lib/mnesia-4.4.7.ez/mnesia-4.4.7/ebin`.

The code server uses module `erl_prim_loader` in ERTS (possibly through `erl_boot_server`) to read code files from archives. However, the functions in `erl_prim_loader` can also be used by other applications to read files from archives. For example, the call `erl_prim_loader:list_dir( "/otp/root/lib/mnesia-4.4.7.ez/mnesia-4.4.7/examples/bench)"` would list the contents of a directory inside an archive. See `erl_prim_loader(3)`.

An application archive file and a regular application directory can coexist. This can be useful when it is needed to have parts of the application as regular files. A typical case is the `priv` directory, which must reside as a regular directory to link in drivers dynamically and start port programs. For other applications that do not need this, directory `priv` can reside in the archive and the files under the directory `priv` can be read through `erl_prim_loader`.

When a directory is added to the code path and when the entire code path is (re)set, the code server decides which subdirectories in an application that are to be read from the archive and which that are to be read as regular files. If directories are added or removed afterwards, the file access can fail if the code path is not updated (possibly to the same path as before, to trigger the directory resolution update).

For each directory on the second level in the application archive (`ebin`, `priv`, `src`, and so on), the code server first chooses the regular directory if it exists and second from the archive. Function `code:lib_dir/2` returns the path to the subdirectory. For example, `code:lib_dir(megaco,ebin)` can return `/otp/root/lib/megaco-3.9.1.1.ez/megaco-3.9.1.1/ebin` while `code:lib_dir(megaco,priv)` can return `/otp/root/lib/megaco-3.9.1.1/priv`. 
When an escript file contains an archive, there are no restrictions on the name of the escript and no restrictions on how many applications that can be stored in the embedded archive. Single Beam files can also reside on the top level in the archive. At startup, the top directory in the embedded archive and all (second level) ebin directories in the embedded archive are added to the code path. See erts:escript(1).

When the choice of directories in the code path is strict, the directory that ends up in the code path is exactly the stated one. This means that if, for example, the directory $OTPROOT/lib/mnesia-4.4.7/ebin is explicitly added to the code path, the code server does not load files from $OTPROOT/lib/mnesia-4.4.7.ez/mnesia-4.4.7/ebin.

This behavior can be controlled through command-line flag `-code_path_choice Choice`. If the flag is set to relaxed, the code server instead chooses a suitable directory depending on the actual file structure. If a regular application ebin directory exists, it is chosen. Otherwise, the directory ebin in the archive is chosen if it exists. If neither of them exists, the original directory is chosen.

Command-line flag `-code_path_choice Choice` also affects how module init interprets the boot script. The interpretation of the explicit code paths in the boot script can be strict or relaxed. It is particularly useful to set the flag to relaxed when elaborating with code loading from archives without editing the boot script. The default is relaxed. See erts:init(3).

Current and Old Code

The code for a module can exist in two variants in a system: current code and old code. When a module is loaded into the system for the first time, the module code becomes 'current' and the global export table is updated with references to all functions exported from the module.

If then a new instance of the module is loaded (for example, because of error correction), the code of the previous instance becomes 'old', and all export entries referring to the previous instance are removed. After that, the new instance is loaded as for the first time, and becomes 'current'.

Both old and current code for a module are valid, and can even be evaluated concurrently. The difference is that exported functions in old code are unavailable. Hence, a global call cannot be made to an exported function in old code, but old code can still be evaluated because of processes lingering in it.

If a third instance of the module is loaded, the code server removes (purges) the old code and any processes lingering in it are terminated. Then the third instance becomes 'current' and the previously current code becomes 'old'.

For more information about old and current code, and how to make a process switch from old to current code, see section Compilation and Code Loading in the Erlang Reference Manual.

Argument Types and Invalid Arguments

Module and application names are atoms, while file and directory names are strings. For backward compatibility reasons, some functions accept both strings and atoms, but a future release will probably only allow the arguments that are documented.

Functions in this module generally fail with an exception if they are passed an incorrect type (for example, an integer or a tuple where an atom is expected). An error tuple is returned if the argument type is correct, but there are some other errors (for example, a non-existing directory is specified to set_path/1).

Error Reasons for Code-Loading Functions

Functions that load code (such as load_file/1) will return {error,Reason} if the load operation fails. Here follows a description of the common reasons.

badfile

The object code has an incorrect format or the module name in the object code is not the expected module name.
code

nofile
  No file with object code was found.

not_purged
  The object code could not be loaded because an old version of
  the code already existed.

on_load_failure
  The module has an -on_load function that failed when it was called.

sticky_directory
  The object code resides in a sticky directory.

Data Types

load_ret() =
  {error, What :: load_error_rsn()} |
  {module, Module :: module()}
load_error_rsn() =
  badfile | nofile | not_purged | on_load_failure | sticky_directory
module_status() = not_loaded | loaded | modified | removed

An opaque term holding prepared code.

Exports

set_path(Path) -> true | {error, What}
Types:
  Path = [Dir :: file:filename()]
  What = bad_directory
Sets the code path to the list of directories Path.
Returns:
  true
    If successful
  {error, bad_directory}
    If any Dir is not a directory name

get_path() -> Path
Types:
  Path = [Dir :: file:filename()]
Returns the code path.

add_path(Dir) -> add_path_ret()
add_pathz(Dir) -> add_path_ret()
Types:
Dir = file:filename()
add_path_ret() = true | \{error, bad_directory\}  

Adds Dir to the code path. The directory is added as the last directory in the new path. If Dir already exists in the path, it is not added.

Returns true if successful, or \{error, bad_directory\} if Dir is not the name of a directory.

add_patha(Dir) -> add_path_ret()

Types:

Dir = file:filename()
add_path_ret() = true | \{error, bad_directory\}

Adds Dir to the beginning of the code path. If Dir exists, it is removed from the old position in the code path.

Returns true if successful, or \{error, bad_directory\} if Dir is not the name of a directory.

add_paths(Dirs) -> ok
add_pathsz(Dirs) -> ok

Types:

Dirs = [Dir :: file:filename()]

Adds the directories in Dirs to the end of the code path. If a Dir exists, it is not added.

Always returns ok, regardless of the validity of each individual Dir.

add_pathsa(Dirs) -> ok

Types:

Dirs = [Dir :: file:filename()]  

Traverses Dirs and adds each Dir to the beginning of the code path. This means that the order of Dirs is reversed in the resulting code path. For example, if you add [Dir1,Dir2], the resulting path will be [Dir2,Dir1 | OldCodePath].

If a Dir already exists in the code path, it is removed from the old position.

Always returns ok, regardless of the validity of each individual Dir.

del_path(NameOrDir) -> boolean() | \{error, What\}

Types:

NameOrDir = Name | Dir  
Name = atom()  
Dir = file:filename()  
What = bad_name

Deletes a directory from the code path. The argument can be an atom Name, in which case the directory with the name .../Name[-Vsn]/[ebin] is deleted from the code path. Also, the complete directory name Dir can be specified as argument.

Returns:

ture

If successful
false
   If the directory is not found
{error, bad_name}
   If the argument is invalid

replace_path(Name, Dir) -> true | {error, What}
Types:
   Name = atom()
   Dir = file:filename()
   What = bad_directory | bad_name | {badarg, term()}
Replaces an old occurrence of a directory named .../Name[-Vsn] [/ebin] in the code path, with Dir. If Name
does not exist, it adds the new directory Dir last in the code path. The new directory must also be named .../Name[-
Vsn] [/ebin]. This function is to be used if a new version of the directory (library) is added to a running system.
Returns:
true
   If successful
{error, bad_name}
   If Name is not found
{error, bad_directory}
   If Dir does not exist
{error, {badarg, [Name, Dir]}}
   If Name or Dir is invalid

load_file(Module) -> load_ret()
Types:
   Module = module()
   load_ret() =
     {error, What :: load_error_rsn()} |
     {module, Module :: module()}
Tries to load the Erlang module Module, using the code path. It looks for the object code file with an extension
corresponding to the Erlang machine used, for example, Module.beam. The loading fails if the module name found
in the object code differs from the name Module.load_binary/3 must be used to load object code with a module
name that is different from the file name.
Returns {module, Module} if successful, or {error, Reason} if loading fails. See Error Reasons for Code-
Loading Functions for a description of the possible error reasons.

load_abs(Filename) -> load_ret()
Types:
   Filename = file:filename()
   load_ret() =
     {error, What :: load_error_rsn()} |
     {module, Module :: module()}
     loaded_filename() =
loaded_ret_atoms() = cover_compiled | preloaded

Same as load_file(Module), but Filename is an absolute or relative filename. The code path is not searched. It returns a value in the same way as load_file/1. Notice that Filename must not contain the extension (for example, .beam) because load_abs/1 adds the correct extension.

equivalent_loaded(Module) -> {module, Module} | {error, What}

Types:
Module = module()
What = module | badfile | nofile | on_load_failure

Tries to load a module in the same way as load_file/1, unless the module is already loaded. However, in embedded mode it does not load a module that is not already loaded, but returns {error, embedded} instead. See Error Reasons for Code-Loading Functions for a description of other possible error reasons.

load_binary(Module, Filename, Binary) ->
{module, Module} | {error, What}

Types:
Module = module()
Filename = loaded_filename()
Binary = binary()
What = badarg | load_error_rsn()
loaded_filename() =
(Filename :: file:filename()) | loaded_ret_atoms()
loaded_ret_atoms() = cover_compiled | preloaded

This function can be used to load object code on remote Erlang nodes. Argument Binary must contain object code for Module. Filename is only used by the code server to keep a record of from which file the object code for Module comes. Thus, Filename is not opened and read by the code server.

Returns {module, Module} if successful, or {error, Reason} if loading fails. See Error Reasons for Code-Loading Functions for a description of the possible error reasons.

atomic_load(Modules) -> ok | {error, [{Module, What}]}{
Types:
Modules = [Module | {Module, Filename, Binary}]
Module = module()
Filename = file:filename()
Binary = binary()
What =
  badfile | nofile | on_load_not_allowed | duplicated |
  not_purged | sticky_directory | pending_on_load

Tries to load all of the modules in the list Modules atomically. That means that either all modules are loaded at the same time, or none of the modules are loaded if there is a problem with any of the modules.

Loading can fail for one the following reasons:
badfile
The object code has an incorrect format or the module name in the object code is not the expected module name.
code

nofile
   No file with object code exists.
on_load_not_allowed
   A module contains an -on_load function.
duplicated
   A module is included more than once in Modules.
not_purged
   The object code cannot be loaded because an old version of the code already exists.
sticky_directory
   The object code resides in a sticky directory.
pending_on_load
   A previously loaded module contains an -on_load function that never finished.

If it is important to minimize the time that an application is inactive while changing code, use prepare_loading/1 and finish_loading/1 instead of atomic_load/1. Here is an example:

```erlang
{ok,Prepared} = code:prepare_loading(Modules),
%% Put the application into an inactive state or do any
%% other preparation needed before changing the code.
ok = code:finish_loading(Prepared),
%% Resume the application.
```

**prepare_loading**([Module | {Module, Filename, Binary}])

Prepares to load the modules in the list `Modules`. Finish the loading by calling `finish_loading(Prepared)`. This function can fail with one of the following error reasons:

**badfile**
   The object code has an incorrect format or the module name in the object code is not the expected module name.

**nofile**
   No file with object code exists.

**on_load_not_allowed**
   A module contains an -on_load function.

**duplicated**
   A module is included more than once in `Modules`.  

<table>
<thead>
<tr>
<th>Types:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modules = [Module</td>
</tr>
<tr>
<td>Module = module()</td>
</tr>
<tr>
<td>Filename = file:filename()</td>
</tr>
<tr>
<td>Binary = binary()</td>
</tr>
<tr>
<td>Prepared = prepared_code()</td>
</tr>
<tr>
<td>What = badfile</td>
</tr>
</tbody>
</table>

66 | Ericsson AB. All Rights Reserved.: Kernel
finish_loading(Prepared) -> ok | {error, [{Module, What}]}  
Types:  
  Prepared = prepared_code()  
  Module = module()  
  What = not_purged | sticky_directory | pending_on_load  

Tries to load code for all modules that have been previously prepared by prepare_loading/1. The loading occurs atomically, meaning that either all modules are loaded at the same time, or none of the modules are loaded.  

This function can fail with one of the following error reasons:  

not_purged  
  The object code cannot be loaded because an old version of the code already exists.  

sticky_directory  
  The object code resides in a sticky directory.  

pending_on_load  
  A previously loaded module contains an -on_load function that never finished.  

discard(Modules :: [Module]) -> ok | {error, [{Module, Reason}]}  
Types:  
  Module = module()  
  What = badfile | nofile | on_load_failure  

Tries to load any modules not already loaded in the list Modules in the same way as load_file/1.  

Returns ok if successful, or {error, [{Module, Reason}]} if loading of some modules fails. See Error Reasons for Code-Loading Functions for a description of other possible error reasons.  

dead(Module) -> boolean()  
Types:  
  Module = module()  

Removes the current code for Module, that is, the current code for Module is made old. This means that processes can continue to execute the code in the module, but no external function calls can be made to it.  

Returns true if successful, or false if there is old code for Module that must be purged first, or if Module is not a (loaded) module.  

purge(Module) -> boolean()  
Types:  
  Module = module()  

Purges the code for Module, that is, removes code marked as old. If some processes still linger in the old code, these processes are killed before the code is removed.  

Note:  
As of ERTS version 9.0, a process is only considered to be lingering in the code if it has direct references to the code. For more information see documentation of erlang:check_process_code/3, which is used in order to determine this.
Returns `true` if successful and any process is needed to be killed, otherwise `false`.

**soft_purge(Module) -> boolean()**

Types:
   - `Module` = `module()`

Purges the code for `Module`, that is, removes code marked as old, but only if no processes linger in it.

**Note:**
As of ERTS version 9.0, a process is only considered to be lingering in the code if it has direct references to the code. For more information see documentation of `erlang:check_process_code/3`, which is used in order to determine this.

Returns `false` if the module cannot be purged because of processes lingering in old code, otherwise `true`.

**is_loaded(Module) -> {file, Loaded} | false**

Types:
   - `Module` = `module()`
     - `Loaded` = `loaded_filename()`

`loaded_filename()` =
   - `(Filename :: file:filename()) | loaded_ret_atoms()`

  `Filename` is an absolute filename.

`loaded_ret_atoms()` = `cover_compiled` | `preloaded`

Checks if `Module` is loaded. If it is, `{file, Loaded}` is returned, otherwise `false`.

Normally, `Loaded` is the absolute filename `Filename` from which the code is obtained. If the module is preloaded (see `script(4)`), `Loaded==preloaded`. If the module is Cover-compiled (see `cover(3)`), `Loaded==cover_compiled`.

**all_available() -> [{Module, Filename, Loaded}]**

Types:
   - `Module` = `string()`
   - `Filename` = `loaded_filename()`
     - `Loaded` = `boolean()`

`loaded_filename()` =
   - `(Filename :: file:filename()) | loaded_ret_atoms()`

  `Filename` is an absolute filename.

`loaded_ret_atoms()` = `cover_compiled` | `preloaded`

Returns a list of tuples `{Module, Filename, Loaded}` for all available modules. A module is considered to be available if it either is loaded or would be loaded if called. `Filename` is normally the absolute filename, as described for `is_loaded/1`.

**all_loaded() -> [{Module, Loaded}]**

Types:
Module = module()
Loaded = loaded_filename()
loaded_filename() =
   (Filename :: file:filename()) | loaded_ret_atoms()
Filename is an absolute filename.
loaded_ret_atoms() = cover_compiled | preloaded

Returns a list of tuples {Module, Loaded} for all loaded modules. Loaded is normally the absolute filename, as described for is_loaded/1.

which(Module) -> Which
Types:
   Module = module()
   Which = loaded_filename() | non_exisiting
loaded_filename() =
   (Filename :: file:filename()) | loaded_ret_atoms()
loaded_ret_atoms() = cover_compiled | preloaded

If the module is not loaded, this function searches the code path for the first file containing object code for Module and returns the absolute filename.

get_object_code(Module) -> {Module, Binary, Filename} | error
Types:
   Module = module()
   Binary = binary()
   Filename = file:filename()

Searches the code path for the object code of module Module. Returns {Module, Binary, Filename} if successful, otherwise error. Binary is a binary data object, which contains the object code for the module. This can be useful if code is to be loaded on a remote node in a distributed system. For example, loading module Module on a node Node is done as follows:

```erlang
...{Module, Binary, Filename} = code:get_object_code(Module),
 rpc:call(Node, code, load_binary, [Module, Filename, Binary]),
...```

get_doc(Mod) -> {ok, Res} | {error, Reason}
Types:
Mod = module()
Res = #docs_v1{}
Reason = non_existing | missing | file:posix()

Searches the code path for EEP-48 style documentation and returns it if available. If no documentation can be found the function tries to generate documentation from the debug information in the module. If no debug information is available, this function will return {error,missing}.

For more information about the documentation chunk see Documentation Storage in Erl_Docgen's User's Guide.

root_dir() -> file:filename()
Returns the root directory of Erlang/OTP, which is the directory where it is installed.

Example:

goto:root_dir().
"/usr/local/otp"

lib_dir() -> file:filename()
Returns the library directory, $OTPROOT/lib, where $OTPROOT is the root directory of Erlang/OTP.

Example:

goto:lib_dir().
"/usr/local/otp/lib"

lib_dir(Name) -> file:filename() | {error, bad_name}
Types:
Name = atom()

Returns the path for the "library directory", the top directory, for an application Name located under $OTPROOT/lib or on a directory referred to with environment variable ERL_LIBS.

If a regular directory called Name or Name-Vsn exists in the code path with an ebin subdirectory, the path to this directory is returned (not the ebin directory).

If the directory refers to a directory in an archive, the archive name is stripped away before the path is returned. For example, if directory /usr/local/otp/lib/mnesia-4.2.2.ez/mnesia-4.2.2/ebin is in the path, /usr/local/otp/lib/mnesia-4.2.2/ebin is returned. This means that the library directory for an application is the same, regardless if the application resides in an archive or not.

Example:

goto:lib_dir(mnesia).
"/usr/local/otp/lib/mnesia-4.2.2"

Returns {error, bad_name} if Name is not the name of an application under $OTPROOT/lib or on a directory referred to through environment variable ERL_LIBS. Fails with an exception if Name has the wrong type.

### Warning:
For backward compatibility, Name is also allowed to be a string. That will probably change in a future release.
lib_dir(Name, SubDir) -> file:filename() | {error, bad_name}
Types:
  Name = SubDir = atom()

Returns the path to a subdirectory directly under the top directory of an application. Normally the subdirectories reside under the top directory for the application, but when applications at least partly resides in an archive, the situation is different. Some of the subdirectories can reside as regular directories while other reside in an archive file. It is not checked whether this directory exists.

Example:

```
> code:lib_dir(megaco, priv).
"/usr/local/otp/lib/megaco-3.9.1.1/priv"
```

Fails with an exception if Name or SubDir has the wrong type.

compiler_dir() -> file:filename()

Returns the compiler library directory. Equivalent to code:lib_dir(compiler).

priv_dir(Name) -> file:filename() | {error, bad_name}
Types:
  Name = atom()

Returns the path to the priv directory in an application. Equivalent to code:lib_dir(Name, priv).

Warning:
For backward compatibility, Name is also allowed to be a string. That will probably change in a future release.

objfile_extension() -> nonempty_string()

Returns the object code file extension corresponding to the Erlang machine used, namely .beam.

stick_dir(Dir) -> ok | error
Types:
  Dir = file:filename()
Marks Dir as sticky.
Returns ok if successful, otherwise error.

unstick_dir(Dir) -> ok | error
Types:
  Dir = file:filename()
Unsticks a directory that is marked as sticky.
Returns ok if successful, otherwise error.

is_sticky(Module) -> boolean()
Types:
Module = module()
Returns true if Module is the name of a module that has been loaded from a sticky directory (in other words: an attempt to reload the module will fail), or false if Module is not a loaded module or is not sticky.

where_is_file(Filename) -> non_existing | Absname
Types:
    Filename = Absname = file:filename()
Searches the code path for Filename, a file of arbitrary type. If found, the full name is returned. non_existing is returned if the file cannot be found. The function can be useful, for example, to locate application resource files.

clash() -> ok
Searches all directories in the code path for module names with identical names and writes a report to stdout.

module_status() -> [{module(), module_status()}]
Types:
    module_status() = not_loaded | loaded | modified | removed
See module_status/1 and all_loaded/0 for details.

module_status(Module :: module() | [module()]) -> module_status() | [{module(), module_status()}]
Types:
    module_status() = not_loaded | loaded | modified | removed
The status of a module can be one of:
not_loaded
    If Module is not currently loaded.
loaded
    If Module is loaded and the object file exists and contains the same code.
removed
    If Module is loaded but no corresponding object file can be found in the code path.
modified
    If Module is loaded but the object file contains code with a different MD5 checksum.
Preloaded modules are always reported as loaded, without inspecting the contents on disk. Cover compiled modules will always be reported as modified if an object file exists, or as removed otherwise. Modules whose load path is an empty string (which is the convention for auto-generated code) will only be reported as loaded or not_loaded. For modules that have native code loaded (see is_module_native/1), the MD5 sum of the native code in the object file is used for the comparison, if it exists; the Beam code in the file is ignored. Reversely, for modules that do not currently have native code loaded, any native code in the file will be ignored.
See also modified_modules/0.

modified_modules() -> [module()]
Returns the list of all currently loaded modules for which module_status/1 returns modified. See also all_loaded/0.
is_module_native(Module) -> true | false | undefined

Types:
    Module = module()

Returns:
true
    If Module is the name of a loaded module that has native code loaded
false
    If Module is loaded but does not have native code
undefined
    If Module is not loaded

get_mode() -> embedded | interactive

Returns an atom describing the mode of the code server: interactive or embedded.

This information is useful when an external entity (for example, an IDE) provides additional code for a running node. If the code server is in interactive mode, it only has to add the path to the code. If the code server is in embedded mode, the code must be loaded with load_binary/3.
A **configuration file** contains values for configuration parameters for the applications in the system. The `erl` command-line argument `-config Name` tells the system to use data in the system configuration file `Name.config`. Configuration parameter values in the configuration file override the values in the application resource files (see `app(4)`). The values in the configuration file can be overridden by command-line flags (see `erts:erl(1)`). The value of a configuration parameter is retrieved by calling `application:get_env/1,2`.

**File Syntax**

The configuration file is to be called `Name.config`, where `Name` is any name. `File.config` contains a single Erlang term and has the following syntax:

```
[[Application1, [{Par11, Val11}, ...]],
 ...  
{ApplicationN, [{ParN1, ValN1}, ...]]].
```

- **Application** = `atom()`
  - Application name.
- **Par** = `atom()`
  - Name of a configuration parameter.
- **Val** = `term()`
  - Value of a configuration parameter.

**sys.config**

When starting Erlang in embedded mode, it is assumed that exactly one system configuration file is used, named `sys.config`. This file is to be located in `$ROOT/releases/Vsn`, where `$ROOT` is the Erlang/OTP root installation directory and `Vsn` is the release version.

Release handling relies on this assumption. When installing a new release version, the new `sys.config` is read and used to update the application configurations.

This means that specifying another `.config` file, or more `.config` files, leads to inconsistent update of application configurations. There is, however, a syntax for `sys.config` that allows pointing out other `.config` files:

```
{{Application, [{Par, Val}]} | File}.  
```

- **File** = `string()`
  - Name of another `.config` file. Extension `.config` can be omitted. It is recommended to use absolute paths. If a relative path is used, `File` is searched, first, relative from `sys.config` directory, then relative to the current working directory of the emulator, for backward compatibility. This allows to use a `sys.config` pointing out other `.config` files in a release or in a node started manually using `-config ...` with same result whatever the current working directory.

When traversing the contents of `sys.config` and a filename is encountered, its contents are read and merged with the result so far. When an application configuration tuple `{Application, Env}` is found, it is merged with the result so far. Merging means that new parameters are added and existing parameter values overwritten.

**Example:**
sys.config:
[[myapp, [{par1, val1}, {par2, val2}]],
 "~/home/user/myconfig"].

myconfig.config:
[[myapp, [{par2, val3}, {par3, val4}]}].

This yields the following environment for myapp:

[[par1, val1], [par2, val3], [par3, val4]]

The behavior if a file specified in sys.config does not exist, or is erroneous, is backwards compatible. Starting the runtime system will fail. Installing a new release version will not fail, but an error message is returned and the erroneous file is ignored.

See Also

app(4), erts:erl(1), OTP Design Principles
disk_log

Erlang module

disk_log is a disk-based term logger that enables efficient logging of items on files.

Two types of logs are supported:

halt logs

Appends items to a single file, which size can be limited by the disk_log module.

wrap logs

Uses a sequence of wrap log files of limited size. As a wrap log file is filled up, further items are logged on to the next file in the sequence, starting all over with the first file when the last file is filled up.

For efficiency reasons, items are always written to files as binaries.

Two formats of the log files are supported:

internal format

Supports automatic repair of log files that are not properly closed and enables efficient reading of logged items in chunks using a set of functions defined in this module. This is the only way to read internally formatted logs.

An item logged to an internally formatted log must not occupy more than 4 GB of disk space (the size must fit in 4 bytes).

external format

Leaves it up to the user to read and interpret the logged data. The disk_log module cannot repair externally formatted logs.

For each open disk log, one process handles requests made to the disk log. This process is created when open/1 is called, provided there exists no process handling the disk log. A process that opens a disk log can be an owner or an anonymous user of the disk log. Each owner is linked to the disk log process, and an owner can close the disk log either explicitly (by calling close/1 or lclose/1,2) or by terminating.

Owners can subscribe to notifications, messages of the form {disk_log, Node, Log, Info}, which are sent from the disk log process when certain events occur, see the functions and in particular the open/1 option notify. A log can have many owners, but a process cannot own a log more than once. However, the same process can open the log as a user more than once.

For a disk log process to close its file properly and terminate, it must be closed by its owners and once by some non-owner process for each time the log was used anonymously. The users are counted and there must not be any users left when the disk log process terminates.

Items can be logged synchronously by using functions log/2, blog/2, log_terms/2, and blog_terms/2. For each of these functions, the caller is put on hold until the items are logged (but not necessarily written, use sync/1 to ensure that). By adding an a to each of the mentioned function names, we get functions that log items asynchronously. Asynchronous functions do not wait for the disk log process to write the items to the file, but return the control to the caller more or less immediately.

When using the internal format for logs, use functions log/2, log_terms/2, alog/2, and alog_terms/2. These functions log one or more Erlang terms. By prefixing each of the functions with a b (for "binary"), we get the corresponding blog() functions for the external format. These functions log one or more chunks of bytes. For example, to log the string "hello" in ASCII format, you can use disk_log:blog(Log, "hello"), or disk_log:blog(Log, list_to_binary("hello")). The two alternatives are equally efficient.

The blog() functions can also be used for internally formatted logs, but in this case they must be called with binaries constructed with calls to term_to_binary/1. There is no check to ensure this, it is entirely the responsibility of
the caller. If these functions are called with binaries that do not correspond to Erlang terms, the chunk/2,3 and automatic repair functions fail. The corresponding terms (not the binaries) are returned when chunk/2,3 is called.

**Note:**
The distributed disk log feature has been deprecated. This feature has also been scheduled for removal in OTP 24.

A collection of open disk logs with the same name running on different nodes is said to be a **distributed disk log** if requests made to any of the logs are automatically made to the other logs as well. The members of such a collection are called individual distributed disk logs, or just distributed disk logs if there is no risk of confusion. There is no order between the members of such a collection. For example, logged terms are not necessarily written to the node where the request was made before written to the other nodes. However, a few functions do not make requests to all members of distributed disk logs, namely info/1,chunk/2,3,bchunk/2,3,chunk_step/3, and lclose/1,2.

An open disk log that is not a distributed disk log is said to be a **local disk log**. A local disk log is only accessible from the node where the disk log process runs, whereas a distributed disk log is accessible from all nodes in the Erlang system, except for those nodes where a local disk log with the same name as the distributed disk log exists. All processes on nodes that have access to a local or distributed disk log can log items or otherwise change, inspect, or close the log.

It is not guaranteed that all log files of a distributed disk log contain the same log items. No attempt is made to synchronize the contents of the files. However, as long as at least one of the involved nodes is alive at each time, all items are logged. When logging items to a distributed log, or otherwise trying to change the log, the replies from individual logs are ignored. If all nodes are down, the disk log functions reply with a **nonode** error.

**Note:**
In some applications, it can be unacceptable that replies from individual logs are ignored. An alternative in such situations is to use many local disk logs instead of one distributed disk log, and implement the distribution without use of the disk_log module.

Errors are reported differently for asynchronous log attempts and other uses of the disk_log module. When used synchronously, this module replies with an error message, but when called asynchronously, this module does not know where to send the error message. Instead, owners subscribing to notifications receive an **error_status** message.

The disk_log module does not report errors to the error_logger module. It is up to the caller to decide whether to employ the error logger. Function format_error/1 can be used to produce readable messages from error replies. However, information events are sent to the error logger in two situations, namely when a log is repaired, or when a file is missing while reading chunks.

Error message **no_such_log** means that the specified disk log is not open. Nothing is said about whether the disk log files exist or not.

**Note:**
If an attempt to reopen or truncate a log fails (see reopen/2,3 and truncate/1,2) the disk log process terminates immediately. Before the process terminates, links to owners and blocking processes (see block/1,2) are removed. The effect is that the links work in one direction only. Any process using a disk log must check for error message **no_such_log** if some other process truncates or reopens the log simultaneously.

**Data Types**

```erlang
log() = term()
dlog_size() =
  infinity |
```

Ericsson AB. All Rights Reserved.: Kernel | 77
**disk_log**

```
integer() >= 1 | 
{MaxNoBytes :: integer() >= 1, MaxNoFiles :: integer() >= 1}

dlog_format() = external | internal
dlog_head_opt() = none | term() | iodata()
dlog_mode() = read_only | read_write
dlog_type() = halt | wrap
continuation()

Chunk continuation returned by chunk/2,3, bchunk/2,3, or chunk_step/3.

invalid_header() = term()
file_error() = term()

Exports

accessible_logs() -> {[LocalLog], [DistributedLog]}

Types:
   LocalLog = DistributedLog = log()

Returns the names of the disk logs accessible on the current node. The first list contains local disk logs and the second list contains distributed disk logs.

alog(Log, Term) -> notify_ret()
balog(Log, Bytes) -> notify_ret()

Types:
   Log = log()
   Term = term()
   Bytes = iodata()
   notify_ret() = ok | {error, no_such_log}

Asynchronously append an item to a disk log. alog/2 is used for internally formatted logs and balog/2 for externally formatted logs. balog/2 can also be used for internally formatted logs if the binary is constructed with a call to term_to_binary/1.

Owners subscribing to notifications receive message read_only, blocked_log, or format_external if the item cannot be written on the log, and possibly one of the messages wrap, full, or error_status if an item is written on the log. Message error_status is sent if something is wrong with the header function or if a file error occurs.

alog_terms(Log, TermList) -> notify_ret()
balog_terms(Log, ByteList) -> notify_ret()

Types:
   Log = log()
   TermList = [term()]
   ByteList = [iodata()]
   notify_ret() = ok | {error, no_such_log}

Asynchronously append a list of items to a disk log. alog_terms/2 is used for internally formatted logs and balog_terms/2 for externally formatted logs. balog_terms/2 can also be used for internally formatted logs if the binaries are constructed with calls to term_to_binary/1.

78 | Ericsson AB. All Rights Reserved.: Kernel
Owners subscribing to notifications receive message read_only, blocked_log, or format_external if the items cannot be written on the log, and possibly one or more of the messages wrap, full, and error_status if items are written on the log. Message error_status is sent if something is wrong with the header function or if a file error occurs.

block(Log) -> ok | {error, block_error_rsn()}
block(Log, QueueLogRecords) -> ok | {error, block_error_rsn()}

Types:
  Log = log()
  QueueLogRecords = boolean()
  block_error_rsn() = no_such_log | nonode | {blocked_log, log()}

With a call to block/1,2 a process can block a log. If the blocking process is not an owner of the log, a temporary link is created between the disk log process and the blocking process. The link ensures that the disk log is unblocked if the blocking process terminates without first closing or unblocking the log.

Any process can probe a blocked log with info/1 or close it with close/1. The blocking process can also use functions chunk/2,3, bcchunk/2,3, chunk_step/3, and unblock/1 without being affected by the block. Any other attempt than those mentioned so far to update or read a blocked log suspends the calling process until the log is unblocked or returns error message {blocked_log, Log}, depending on whether the value of QueueLogRecords is true or false. QueueLogRecords defaults to true, which is used by block/1.

change_header(Log, Header) -> ok | {error, Reason}

Types:
  Log = log()
  Header =
    {head, dlog_head_opt()} |
    {head_func, MFA :: {atom(), atom(), list()}}
  Reason =
    no_such_log | nonode |
    {read_only_mode, Log} |
    {blocked_log, Log} |
    {badarg, head}

Changes the value of option head or head_func for an owner of a disk log.

change_notify(Log, Owner, Notify) -> ok | {error, Reason}

Types:
  Log = log()
  Owner = pid()
  Notify = boolean()
  Reason =
    no_such_log | nonode |
    {blocked_log, Log} |
    {badarg, notify} |
    {not_owner, Owner}

Changes the value of option notify for an owner of a disk log.
change_size(Log, Size) -> ok | error, Reason

Types:
   Log = log()
   Size = dlog_size()
   Reason =
      no_such_log | nonode |
      {read_only_mode, Log} |
      {blocked_log, Log} |
      {new_size_too_small, Log, CurrentSize :: integer() >= 1} |
      {badarg, size} |
      {file_error, file:filename(), file_error()}

Changes the size of an open log. For a halt log, the size can always be increased, but it cannot be decreased to something less than the current file size.

For a wrap log, both the size and the number of files can always be increased, as long as the number of files does not exceed 65000. If the maximum number of files is decreased, the change is not valid until the current file is full and the log wraps to the next file. The redundant files are removed the next time the log wraps around, that is, starts to log to file number 1.

As an example, assume that the old maximum number of files is 10 and that the new maximum number of files is 6. If the current file number is not greater than the new maximum number of files, files 7-10 are removed when file 6 is full and the log starts to write to file number 1 again. Otherwise, the files greater than the current file are removed when the current file is full (for example, if the current file is 8, files 9 and 10 are removed). The files between the new maximum number of files and the current file (that is, files 7 and 8) are removed the next time file 6 is full.

If the size of the files is decreased, the change immediately affects the current log. It does not change the size of log files already full until the next time they are used.

If the log size is decreased, for example, to save space, function inc_wrap_file/1 can be used to force the log to wrap.

chunk(Log, Continuation) -> chunk_ret()
chunk(Log, Continuation, N) -> chunk_ret()
bchunk(Log, Continuation) -> bchunk_ret()
bchunk(Log, Continuation, N) -> bchunk_ret()

Types:
   Log = log()
   Continuation = start | continuation()
   N = integer() >= 1 | infinity
   chunk_ret() =
      {Continuation2 :: continuation(), Terms :: [term()]} |
      {Continuation2 :: continuation(),
       Terms :: [term()],
       Badbytes :: integer() >= 0} |
      eof |
      {error, Reason :: chunk_error_rsn()}
   bchunk_ret() =
      {Continuation2 :: continuation(), Binaries :: [binary()]} |
      {Continuation2 :: continuation(),
       Binaries :: [binary()],
       Badbytes :: integer() >= 0} |
Efficiently reads the terms that are appended to an internally formatted log. It minimizes disk I/O by reading 64 kilobyte chunks from the file. Functions bchunk/2,3 return the binaries read from the file, they do not call binary_to_term(). Apart from that, they work just like chunk/2,3.

The first time chunk() (or bchunk()) is called, an initial continuation, the atom start, must be provided. If a disk log process is running on the current node, terms are read from that log. Otherwise, an individual distributed log on some other node is chosen, if such a log exists.

When chunk/3 is called, N controls the maximum number of terms that are read from the log in each chunk. Defaults to infinity, which means that all the terms contained in the 64 kilobyte chunk are read. If less than N terms are returned, this does not necessarily mean that the end of the file is reached.

chunk() returns a tuple {Continuation2, Terms}, where Terms is a list of terms found in the log. Continuation2 is yet another continuation, which must be passed on to any subsequent calls to chunk(). With a series of calls to chunk(), all terms from a log can be extracted.

chunk() returns a tuple {Continuation2, Terms, Badbytes} if the log is opened in read-only mode and the read chunk is corrupt. Badbytes is the number of bytes in the file found not to be Erlang terms in the chunk. Notice that the log is not repaired. When trying to read chunks from a log opened in read-write mode, tuple {corrupt_log_file, FileName} is returned if the read chunk is corrupt.

chunk() returns eof when the end of the log is reached, or {error, Reason} if an error occurs. If a wrap log file is missing, a message is output on the error log.

When chunk/2,3 is used with wrap logs, the returned continuation might not be valid in the next call to chunk(). This is because the log can wrap and delete the file into which the continuation points. To prevent this, the log can be blocked during the search.

chunk_info(Continuation) -> InfoList | {error, Reason}

Types:
  Continuation = continuation()
  InfoList = [{node, Node :: node()}, ...]
  Reason = {no_continuation, Continuation}

Returns the pair {node, Node}, describing the chunk continuation returned by chunk/2,3, bchunk/2,3, or chunk_step/3.

Terms are read from the disk log running on Node.

chunk_step(Log, Continuation, Step) ->
  {ok, any()} | {error, Reason}

Types:
Log = log()
Continuation = start | continuation()
Step = integer()
Reason =
    no_such_log | end_of_log |
file_error, file:filename(), file_error()
Can be used with \texttt{chunk/2,3} and \texttt{bchunk/2,3} to search through an internally formatted wrap log. It takes as argument a continuation as returned by \texttt{chunk/2,3}, \texttt{bchunk/2,3} or \texttt{chunk_step/3} and steps forward (or backward) \texttt{Step} files in the wrap log. The continuation returned, points to the first log item in the new current file.
If atom \texttt{start} is specified as continuation, a disk log to read terms from is chosen. A local or distributed disk log on the current node is preferred to an individual distributed log on some other node.
If the wrap log is not full because all files are not yet used, \{error, end_of_log\} is returned if trying to step outside the log.

close(Log) -> ok | \{error, close_error_rsn()\}
Types:
    Log = log()
    close_error_rsn() =
    no_such_log | nonode | 
    file_error, file:filename(), file_error()
Closes a local or distributed disk log properly. An internally formatted log must be closed before the Erlang system is stopped. Otherwise, the log is regarded as unclosed and the automatic repair procedure is activated next time the log is opened.
The disk log process is not terminated as long as there are owners or users of the log. All owners must close the log, possibly by terminating. Also, any other process, not only the processes that have opened the log anonymously, can decrement the \texttt{users} counter by closing the log. Attempts to close a log by a process that is not an owner are ignored if there are no users.
If the log is blocked by the closing process, the log is also unblocked.

format_error(Error) -> io_lib:chars()
Types:
    Error = term()
Given the error returned by any function in this module, this function returns a descriptive string of the error in English. For file errors, function \texttt{format_error/1} in module \texttt{file} is called.

inc_wrap_file(Log) -> ok | \{error, inc_wrap_error_rsn()\}
Types:
    Log = log()
    inc_wrap_error_rsn() =
    no_such_log | nonode | 
    file_error, file:filename(), file_error()
{invalid_header, invalid_header()} | {file_error, file:filename(), file_error()}
invalid_header() = term()

Forces the internally formatted disk log to start logging to the next log file. It can be used, for example, with change_size/2 to reduce the amount of disk space allocated by the disk log.

Owners subscribing to notifications normally receive a \texttt{wrap} message, but if an error occurs with a reason tag of \texttt{invalid_header} or \texttt{file_error}, an \texttt{error_status} message is sent.

\[
\text{info(Log) -> InfoList | \{error, no_such_log\}}
\]

Types:
\[
Log = \log() \quad \text{InfoList = [dlog_info()]}
\]

\[
dlog_info() =
\text{\{name, Log :: log()\}} | \\
\text{\{file, File :: file:filename()\}} | \\
\text{\{type, Type :: dlog_type()\}} | \\
\text{\{format, Format :: dlog_format()\}} | \\
\text{\{size, Size :: dlog_size()\}} | \\
\text{\{mode, Mode :: dlog_mode()\}} | \\
\text{\{owners, [[\{pid(), Notify :: boolean()\}]\]]\} | \\
\text{\{users, Users :: integer() >= 0\}} | \\
\text{\{status, Status :: ok | \{blocked, QueueLogRecords :: boolean()\}\]} | \\
\text{\{node, Node :: node()\}} | \\
\text{\{distributed, Dist :: local | [node()]\}} | \\
\text{\{head, Head :: none | \{head, term()\} | (MFA :: \{atom(), atom(), list()\}\)}} | \\
\text{\{no_written_items, NoWrittenItems :: integer() >= 0\}} | \\
\text{\{full, Full :: boolean\}} | \\
\text{\{no_current_bytes, integer() >= 0\}} | \\
\text{\{no_current_items, integer() >= 0\}} | \\
\text{\{no_items, integer() >= 0\}} | \\
\text{\{current_file, integer() >= 1\}} | \\
\text{\{no_overflows, \\
\text{\{SinceLogWasOpened :: integer() >= 0,} \\
\text{\{SinceLastInfo :: integer() >= 0\}\}}}}
\]

Returns a list of \{(Tag, Value)\} pairs describing the log. If a disk log process is running on the current node, that log is used as source of information, otherwise an individual distributed log on some other node is chosen, if such a log exists.

The following pairs are returned for all logs:

\{(name, Log)\}

Log is the log name as specified by the open/1 option name.

\{(file, File)\}

For halt logs File is the filename, and for wrap logs File is the base name.
{type, Type}

Type is the log type as specified by the open/1 option type.

{format, Format}

Format is the log format as specified by the open/1 option format.

{size, Size}

Size is the log size as specified by the open/1 option size, or the size set by change_size/2. The value set by change_size/2 is reflected immediately.

{mode, Mode}

Mode is the log mode as specified by the open/1 option mode.

{owners, [{pid(), Notify}]}  

Notify is the value set by the open/1 option notify or function change_notify/3 for the owners of the log.

{users, Users}

Users is the number of anonymous users of the log, see the open/1 option linkto.

{status, Status}

Status is ok or {blocked, QueueLogRecords} as set by functions block/1,2 and unblock/1.

{node, Node}

The information returned by the current invocation of function info/1 is gathered from the disk log process running on Node.

{distributed, Dist}

If the log is local on the current node, Dist has the value local, otherwise all nodes where the log is distributed are returned as a list.

**Warning:**

The distributed disk log feature has been deprecated. This feature has also been scheduled for removal in OTP 24.

The following pairs are returned for all logs opened in read_write mode:

{head, Head}

Depending on the value of the open/1 options head and head_func, or set by function change_header/2, the value of Head is none (default), {head, H} (head option), or {M,F,A} (head_func option).

{no_written_items, NoWrittenItems}

NoWrittenItems is the number of items written to the log since the disk log process was created.

The following pair is returned for halt logs opened in read_write mode:

{full, Full}

Full is true or false depending on whether the halt log is full or not.

The following pairs are returned for wrap logs opened in read_write mode:

{no_current_bytes, integer() >= 0}  

The number of bytes written to the current wrap log file.
The number of items written to the current wrap log file, header inclusive.

The total number of items in all wrap log files.

The ordinal for the current wrap log file in the range 1..MaxNoFiles, where MaxNoFiles is specified by the open/1 option size or set by change_size/2.

SinceLogWasOpened (SinceLastInfo) is the number of times a wrap log file has been filled up and a new one is opened or inc_wrap_file/1 has been called since the disk log was last opened (info/1 was last called). The first time info/2 is called after a log was (re)opened or truncated, the two values are equal.

Notice that functions chunk/2,3, bchunk/2,3, and chunk_step/3 do not affect any value returned by info/1.

\[ \text{lclose(Log)} \rightarrow \text{ok} | \{\text{error, lclose_error_rsn()}} \]
\[ \text{lclose(Log, Node)} \rightarrow \text{ok} | \{\text{error, lclose_error_rsn()}} \]

Types:
\[ \text{Log} = \text{log()}} \]
\[ \text{Node} = \text{node()}} \]
\[ \text{lclose_error_rsn()} = \]
\[ \begin{align*} & \text{no such log} | \{\text{file error, file:filename(), file_error()}} \end{align*} \]

lclose/1 closes a local log or an individual distributed log on the current node.
lclose/2 closes an individual distributed log on the specified node if the node is not the current one.
lclose(Log) is equivalent to lclose(Log, node()). See also close/1.
If no log with the specified name exist on the specified node, no_such_log is returned.

\[ \text{log(Log, Term)} \rightarrow \text{ok} | \{\text{error, Reason :: log_error_rsn()}} \]
\[ \text{blog(Log, Bytes)} \rightarrow \text{ok} | \{\text{error, Reason :: log_error_rsn()}} \]

Types:
\[ \text{Log} = \text{log()}} \]
\[ \text{Term} = \text{term()}} \]
\[ \text{Bytes} = \text{iodata()}} \]
\[ \text{log_error_rsn() =} \]
\[ \begin{align*} & \text{no such log} | \text{nonode} | \]
\[ & \{\text{read only mode, log()}} | \]
\[ & \{\text{format external, log()}} | \]
\[ & \{\text{blocked log, log()}} | \]
\[ & \{\text{full, log()}} | \]
\[ & \{\text{invalid header, invalid header()}} | \]
\[ & \{\text{file error, file:filename(), file_error()}} \]

Synchronously appends a term to a disk log. Returns ok or \{error, Reason\} when the term is written to disk. If the log is distributed, ok is returned, unless all nodes are down. Terms are written by the ordinary write() function of the operating system. Hence, it is not guaranteed that the term is written to disk, it can linger in the operating system kernel for a while. To ensure that the item is written to disk, function sync/1 must be called.
log/2 is used for internally formatted logs, and blog/2 for externally formatted logs. blog/2 can also be used for internally formatted logs if the binary is constructed with a call to term_to_binary/1.

Owners subscribing to notifications are notified of an error with an error_status message if the error reason tag is invalid_header or file_error.

log_terms(Log, TermList) ->
    ok | {error, Reason :: log_error_rsn()}
blog_terms(Log, BytesList) ->
    ok | {error, Reason :: log_error_rsn()}
Types:
    Log = log()
    TermList = [term()]
    BytesList = [iodata()]
log_error_rsn() =
    no_such_log | nonode |
    {read_only_mode, log()} | {format_external, log()} |
    {blocked_log, log()} | {full, log()} |
    {invalid_header, invalid_header()} |
    {file_error, file:filename(), file_error()}

Synchronously appends a list of items to the log. It is more efficient to use these functions instead of functions log/2 and blog/2. The specified list is split into as large sublists as possible (limited by the size of wrap log files), and each sublist is logged as one single item, which reduces the overhead.

log_terms/2 is used for internally formatted logs, and blog_terms/2 for externally formatted logs. blog_terms/2 can also be used for internally formatted logs if the binaries are constructed with calls to term_to_binary/1.

Owners subscribing to notifications are notified of an error with an error_status message if the error reason tag is invalid_header or file_error.

open(ArgL) -> open_ret() | dist_open_ret()
Types:
    ArgL = dlog_options()
    dlog_options() = [dlog_option()]
    dlog_option() =
        {name, Log :: log()} |
        {file, FileName :: file:filename()} |
        {linkto, LinkTo :: none | pid()} |
        {repair, Repair :: true | false | truncate} |
        {type, Type :: dlog_type()} |
        {format, Format :: dlog_format()} |
        {size, Size :: dlog_size()} |
        {distributed, Nodes :: [node()]} |
        {notify, boolean()} |
        {head, Head :: dlog_head_opt()} |
        {head_func, MFA :: {atom(), atom(), list()}} |
        {quiet, boolean()}

86 | Ericsson AB. All Rights Reserved.: Kernel
{mode, Mode :: dlog_mode()}

open_ret() = ret() | {error, open_error_rsn()}

ret() =
  {ok, Log :: log()}
  {repaired, Log :: log(),
   {recovered, Rec :: integer() >= 0},
   {badbytes, Bad :: integer() >= 0}}

dist_open_ret() =
  [{node(), ret()}, [{node(), {error, dist_error_rsn()}]}]

dist_error_rsn() = nodedown | open_error_rsn()

open_error_rsn() =
  no_such_log |
  {badarg, term()}
  {size_mismatch,
   CurrentSize :: dlog_size(),
   NewSize :: dlog_size()}
  {arg_mismatch,
   OptionName :: dlog_optattr(),
  CurrentValue :: term(),
   Value :: term()}
  {name_already_open, Log :: log()}
  {open_read_write, Log :: log()}
  {need_repair, Log :: log()}
  {not_a_log_file, FileName :: file:filename()}
  {invalid_index_file, FileName :: file:filename()}
  {invalid_header, invalid_header()}
  {file_error, file:filename(), file_error()}
  {node_already_open, Log :: log()}

dlog_optattr() =
  name | file | linkto | repair | type | format | size |
  distributed | notify | head | head_func | mode

dlog_size() =
  infinity |
  integer() >= 1 |
  {MaxNoBytes :: integer() >= 1, MaxNoFiles :: integer() >= 1}

Parameter ArgL is a list of the following options:

{name, Log}

  Specifies the log name. This name must be passed on as a parameter in all subsequent logging operations. A
  name must always be supplied.

{file, FileName}

  Specifies the name of the file to be used for logged terms. If this value is omitted and the log name is an atom or
  a string, the filename defaults to lists:concat([Log, "_.LOG"]) for halt logs.

  For wrap logs, this is the base name of the files. Each file in a wrap log is called <base_name>.N, where N is
  an integer. Each wrap log also has two files called <base_name>.idx and <base_name>.siz.
disk_log

{linkto, LinkTo}

If LinkTo is a pid, it becomes an owner of the log. If LinkTo is none, the log records that it is used anonymously by some process by incrementing the users counter. By default, the process that calls open/1 owns the log.

{repair, Repair}

If Repair is true, the current log file is repaired, if needed. As the restoration is initiated, a message is output on the error log. If false is specified, no automatic repair is attempted. Instead, the tuple {error, {need_repair, Log}} is returned if an attempt is made to open a corrupt log file. If truncate is specified, the log file becomes truncated, creating an empty log. Defaults to true, which has no effect on logs opened in read-only mode.

{type, Type}

The log type. Defaults to halt.

{format, Format}

Disk log format. Defaults to internal.

{size, Size}

Log size.

When a halt log has reached its maximum size, all attempts to log more items are rejected. Defaults to infinity, which for halt implies that there is no maximum size.

For wrap logs, parameter Size can be a pair {MaxNoBytes, MaxNoFiles} or infinity. In the latter case, if the files of an existing wrap log with the same name can be found, the size is read from the existing wrap log, otherwise an error is returned.

Wrap logs write at most MaxNoBytes bytes on each file and use MaxNoFiles files before starting all over with the first wrap log file. Regardless of MaxNoBytes, at least the header (if there is one) and one item are written on each wrap log file before wrapping to the next file.

When opening an existing wrap log, it is not necessary to supply a value for option Size, but any supplied value must equal the current log size, otherwise the tuple {error, {size_mismatch, CurrentSize, NewSize}} is returned.

{distributed, Nodes}

This option can be used for adding members to a distributed disk log. Defaults to [], which means that the log is local on the current node.

Warning:

The distributed disk log feature has been deprecated. This feature has also been scheduled for removal in OTP 24.

{notify, boolean()}

If true, the log owners are notified when certain log events occur. Defaults to false. The owners are sent one of the following messages when an event occurs:

{disk_log, Node, Log, {wrap, NoLostItems}}

Sent when a wrap log has filled up one of its files and a new file is opened. NoLostItems is the number of previously logged items that were lost when truncating existing files.
{disk_log, Node, Log, {truncated, NoLostItems}}

Sent when a log is truncated or reopened. For halt logs NoLostItems is the number of items written on the log since the disk log process was created. For wrap logs NoLostItems is the number of items on all wrap log files.

{disk_log, Node, Log, {read_only, Items}}

Sent when an asynchronous log attempt is made to a log file opened in read-only mode. Items is the items from the log attempt.

{disk_log, Node, Log, {blocked_log, Items}}

Sent when an asynchronous log attempt is made to a blocked log that does not queue log attempts. Items is the items from the log attempt.

{disk_log, Node, Log, {format_external, Items}}

Sent when function alog/2 or alog_terms/2 is used for internally formatted logs. Items is the items from the log attempt.

{disk_log, Node, Log, full}

Sent when an attempt to log items to a wrap log would write more bytes than the limit set by option size.

{disk_log, Node, Log, {error_status, Status}}

Sent when the error status changes. The error status is defined by the outcome of the last attempt to log items to the log, or to truncate the log, or the last use of function sync/1, inc_wrap_file/1, or change_size/2. Status is either ok or {error, Error}, the former is the initial value.

{head, Head}

Specifies a header to be written first on the log file. If the log is a wrap log, the item Head is written first in each new file. Head is to be a term if the format is internal, otherwise a sequence of bytes. Defaults to none, which means that no header is written first on the file.

{head_func, {M,F,A}}

Specifies a function to be called each time a new log file is opened. The call M:F(A) is assumed to return {ok, Head}. The item Head is written first in each file. Head is to be a term if the format is internal, otherwise a sequence of bytes.

{mode, Mode}

Specifies if the log is to be opened in read-only or read-write mode. Defaults to read_write.

{quiet, Boolean}

Specifies if messages will be sent to error_logger on recoverable errors with the log files. Defaults to false.

open/1 returns {ok, Log} if the log file is successfully opened. If the file is successfully repaired, the tuple {repaired, Log, {recovered, Rec}, (badbytes, Bad)} is returned, where Rec is the number of whole Erlang terms found in the file and Bad is the number of bytes in the file that are non-Erlang terms. If the parameter distributed is specified, open/1 returns a list of successful replies and a list of erroneous replies. Each reply is tagged with the node name.

When a disk log is opened in read-write mode, any existing log file is checked for. If there is none, a new empty log is created, otherwise the existing file is opened at the position after the last logged item, and the logging of items starts from there. If the format is internal and the existing file is not recognized as an internally formatted log, a tuple {error, {not_a_log_file, FileName}} is returned.

open/1 cannot be used for changing the values of options of an open log. When there are prior owners or users of a log, all option values except name, linkto, and notify are only checked against the values supplied before as option values to function open/1, change_header/2, change_notify/3, or change_size/2. Thus, none
of the options except name is mandatory. If some specified value differs from the current value, a tuple \(\text{error, \{arg_mismatch, OptionName, currentValue, Value\}}\) is returned.

**Note:**

If an owner attempts to open a log as owner once again, it is acknowledged with the return value \(\text{ok, Log}\), but the state of the disk log is not affected.

If a log with a specified name is local on some node, and one tries to open the log distributed on the same node, the tuple \(\text{error, \{node_already_open, Log\}}\) is returned. The same tuple is returned if the log is distributed on some node, and one tries to open the log locally on the same node. Opening individual distributed disk logs for the first time adds those logs to a (possibly empty) distributed disk log. The supplied option values are used on all nodes mentioned by option distributed. Individual distributed logs know nothing about each other's option values, so each node can be given unique option values by creating a distributed log with many calls to open/1.

A log file can be opened more than once by giving different values to option name or by using the same file when distributing a log on different nodes. It is up to the user of module disk_log to ensure that not more than one disk log process has write access to any file, otherwise the file can be corrupted.

If an attempt to open a log file for the first time fails, the disk log process terminates with the EXIT message \(\{\text{failed, Reason}, \{\text{disk_log, open, 1}\}\}\). The function returns \(\text{error, Reason}\) for all other errors.

\[\text{pid2name}(\text{Pid}) \rightarrow \{\text{ok, Log}\} | \text{undefined} \]

**Types:**

\[\text{Pid} = \text{pid()}\]

\[\text{Log} = \text{log()}\]

Returns the log name given the pid of a disk log process on the current node, or undefined if the specified pid is not a disk log process.

This function is meant to be used for debugging only.

\[\text{reopen}(\text{Log}, \text{File}) \rightarrow \text{ok} | \{\text{error, reopen_error_rsn}\}\]

\[\text{reopen}(\text{Log}, \text{File}, \text{Head}) \rightarrow \text{ok} | \{\text{error, reopen_error_rsn}\}\]

\[\text{breopen}(\text{Log}, \text{File}, \text{BHead}) \rightarrow \text{ok} | \{\text{error, reopen_error_rsn}\}\]

**Types:**

\[\text{Log} = \text{log()}\]

\[\text{File} = \text{file:filename()}\]

\[\text{Head} = \text{term()}\]

\[\text{BHead} = \text{iodata()}\]

\[\text{reopen_error_rsn()} = \]

\[\text{no_such_log} | \text{nonode} | \{\text{read_only_mode, log}\} | \{\text{blocked_log, log}\} | \{\text{same_file_name, log}\} | \{\text{invalid_index_file, file:filename}\} | \{\text{invalid_header, invalid_header}\} | \{\text{file_error, file:filename}, \text{file_error}\}\]

Renames the log file to File and then recreates a new log file. If a wrap log exists, File is used as the base name of the renamed files. By default the header given to open/1 is written first in the newly opened log file, but if argument
Head or BHead is specified, this item is used instead. The header argument is used only once. Next time a wrap log file is opened, the header given to open/1 is used.

reopen/2,3 are used for internally formatted logs, and breopen/3 for externally formatted logs.

Owners subscribing to notifications receive a truncate message.

Upon failure to reopen the log, the disk log process terminates with the EXIT message {{failed,Error}, [{disk_log,Fun,Arity}]). Other processes having requests queued receive the message {disk_log, Node, {error, disk_log_stopped}}.

sync(Log) -> ok | {error, sync_error_rsn()}

Types:

Log = log()
sync_error_rsn() =
  no_such_log | nonode | {read_only_mode, log()} | {blocked_log, log()} | {file_error, file:filename(), file_error()}

Ensures that the contents of the log are written to the disk. This is usually a rather expensive operation.

truncate(Log) -> ok | {error, trunc_error_rsn()}
truncate(Log, Head) -> ok | {error, trunc_error_rsn()}
btruncate(Log, BHead) -> ok | {error, trunc_error_rsn()}

Types:

Log = log()
Head = term()
BHead = iodata()
trunc_error_rsn() =
  no_such_log | nonode | {read_only_mode, log()} | {blocked_log, log()} | {invalid_header, invalid_header()} | {file_error, file:filename(), file_error()}

Removes all items from a disk log. If argument Head or BHead is specified, this item is written first in the newly truncated log, otherwise the header given to open/1 is used. The header argument is used only once. Next time a wrap log file is opened, the header given to open/1 is used.

truncate/1,2 are used for internally formatted logs, and btruncate/2 for externally formatted logs.

Owners subscribing to notifications receive a truncate message.

If the attempt to truncate the log fails, the disk log process terminates with the EXIT message {{failed,Reason}, [{disk_log,Fun,Arity}]). Other processes having requests queued receive the message {disk_log, Node, {error, disk_log_stopped}}.

unblock(Log) -> ok | {error, unblock_error_rsn()}

Types:

Log = log()
unblock_error_rsn() =
  no_such_log | nonode |
Unblocks a log. A log can only be unblocked by the blocking process.

See Also

file(3), pg2(3), wrap_log_reader(3)
erl_boot_server

Erlang module

This server is used to assist diskless Erlang nodes that fetch all Erlang code from another machine.

This server is used to fetch all code, including the start script, if an Erlang runtime system is started with command-line flag `-loader inet`. All hosts specified with command-line flag `-hosts Host` must have one instance of this server running.

This server can be started with the Kernel configuration parameter `start_boot_server`.

The `erl_boot_server` can read regular files and files in archives. See `code(3)` and `erl_prim_loader(3)` in ERTS.

Warning:
The support for loading code from archive files is experimental. It is released before it is ready to obtain early feedback. The file format, semantics, interfaces, and so on, can be changed in a future release.

Exports

add_slave(Slave) -> ok | {error, Reason}
Types:
  Slave = Host
  Host = inet:ip_address() | inet:hostname()
  Reason = {badarg, Slave}
Adds a Slave node to the list of allowed slave hosts.

delete_slave(Slave) -> ok | {error, Reason}
Types:
  Slave = Host
  Host = inet:ip_address() | inet:hostname()
  Reason = {badarg, Slave}
Deletes a Slave node from the list of allowed slave hosts.

start(Slaves) -> {ok, Pid} | {error, Reason}
Types:
  Slaves = [Host]
  Host = inet:ip_address() | inet:hostname()
  Pid = pid()
  Reason = {badarg, Slaves}
Starts the boot server. Slaves is a list of IP addresses for hosts, which are allowed to use this server as a boot server.

start_link(Slaves) -> {ok, Pid} | {error, Reason}
Types:
erl_boot_server

Slaves = [Host]
  Host = inet:ip_address() | inet:hostname()
  Pid = pid()
  Reason = {badarg, Slaves}

Starts the boot server and links to the caller. This function is used to start the server if it is included in a supervision tree.

which_slaves() -> Slaves

Types:
  Slaves = [Slave]
  Slave =
    {Netmask :: inet:ip_address(), Address :: inet:ip_address()}

Returns the current list of allowed slave hosts.

SEE ALSO
erts:init(3), erts:erl_prim_loader(3)
erl_ddll

Erlang module

This module provides an interface for loading and unloading Erlang linked-in drivers in runtime.

**Note:**

This is a large reference document. For casual use of this module, and for most real world applications, the descriptions of functions `load/2` and `unload/1` are enough to get started.

The driver is to be provided as a dynamically linked library in an object code format specific for the platform in use, that is, `.so` files on most Unix systems and `.dll` files on Windows. An Erlang linked-in driver must provide specific interfaces to the emulator, so this module is not designed for loading arbitrary dynamic libraries. For more information about Erlang drivers, see `erts:erl_driver`.

When describing a set of functions (that is, a module, a part of a module, or an application), executing in a process and wanting to use a ddll-driver, we use the term **user**. A process can have many users (different modules needing the same driver) and many processes running the same code, making up many **users** of a driver.

In the basic scenario, each user loads the driver before starting to use it and unloads the driver when done. The reference counting keeps track of processes and the number of loads by each process. This way the driver is only unloaded when no one wants it (it has no user). The driver also keeps track of ports that are opened to it. This enables delay of unloading until all ports are closed, or killing of all ports that use the driver when it is unloaded.

The interface supports two basic scenarios of loading and unloading. Each scenario can also have the option of either killing ports when the driver is unloading, or waiting for the ports to close themselves. The scenarios are as follows:

**Load and Unload on a "When Needed Basis"**

This (most common) scenario simply supports that each user of the driver loads it when needed and unloads it when no longer needed. The driver is always reference counted and as long as a process keeping the driver loaded is still alive, the driver is present in the system.

Each user of the driver use **literally** the same pathname for the driver when demanding load, but the users are not concerned with if the driver is already loaded from the file system or if the object code must be loaded from file system.

The following two pairs of functions support this scenario:

**load/2 and unload/1**

When using the `load/unload` interfaces, the driver is not unloaded until the **last port** using the driver is closed. Function `unload/1` can return immediately, as the users have no interest in when the unloading occurs. The driver is unloaded when no one needs it any longer.

If a process having the driver loaded dies, it has the same effect as if unloading is done.

When loading, function `load/2` returns `ok` when any instance of the driver is present. Thus, if a driver is waiting to get unloaded (because of open ports), it simply changes state to no longer need unloading.

**load_driver/2 and unload_driver/1**

These interfaces are intended to be used when it is considered an error that ports are open to a driver that no user has loaded. The ports that are still open when the last user calls `unload_driver/1` or when the last process having the driver loaded dies, are killed with reason `driver_unloaded`.

The function names `load_driver` and `unload_driver` are kept for backward compatibility.
**Loading and Reloading for Code Replacement**

This scenario can occur if the driver code needs replacement during operation of the Erlang emulator. Implementing driver code replacement is a little more tedious than Beam code replacement, as one driver cannot be loaded as both "old" and "new" code. All users of a driver must have it closed (no open ports) before the old code can be unloaded and the new code can be loaded.

The unloading/loading is done as one atomic operation, blocking all processes in the system from using the driver in question while in progress.

The preferred way to do driver code replacement is to let one single process keep track of the driver. When the process starts, the driver is loaded. When replacement is required, the driver is reloaded. Unload is probably never done, or done when the process exits. If more than one user has a driver loaded when code replacement is demanded, the replacement cannot occur until the last "other" user has unloaded the driver.

Demanding reload when a reload is already in progress is always an error. Using the high-level functions, it is also an error to demand reloading when more than one user has the driver loaded.

To simplify driver replacement, avoid designing your system so that more than one user has the driver loaded.

The two functions for reloading drivers are to be used together with corresponding load functions to support the two different behaviors concerning open ports:

**load/2 and reload/2**

This pair of functions is used when reloading is to be done after the last open port to the driver is closed.

As reload/2 waits for the reloading to occur, a misbehaving process keeping open ports to the driver (or keeping the driver loaded) can cause infinite waiting for reload. Time-outs must be provided outside of the process demanding the reload or by using the low-level interface try_load/3 in combination with driver monitors.

**load_driver/2 and reload_driver/2**

This pair of functions are used when open ports to the driver are to be killed with reason driver_unloaded to allow for new driver code to get loaded.

However, if another process has the driver loaded, calling reload_driver returns error code pending_process. As stated earlier, the recommended design is to not allow other users than the "driver reloader" to demand loading of the driver in question.

**Data Types**

driver() = iolist() | atom()
pool() = string() | atom()

**Exports**

demonitor(MonitorRef) -> ok

Types:

MonitorRef = reference()

Removes a driver monitor in much the same way as erlang:demonitor/1 in ERTS does with process monitors. For details about how to create driver monitors, see monitor/2, try_load/3, and try_unload/2.

The function throws a badarg exception if the parameter is not a reference().

format_error(ErrorDesc) -> string()

Types:
ErrorDesc = term()

Takes an ErrorDesc returned by load, unload, or reload functions and returns a string that describes the error or warning.

**Note:**

Because of peculiarities in the dynamic loading interfaces on different platforms, the returned string is only guaranteed to describe the correct error if format_error/1 is called in the same instance of the Erlang virtual machine as the error appeared in (meaning the same operating system process).

```erlang
info() -> AllInfoList
Types:
  AllInfoList = [DriverInfo]
  DriverInfo = {DriverName, InfoList}
  DriverName = string()
  InfoList = [InfoItem]
  InfoItem = {Tag :: atom(), Value :: term()}

Returns a list of tuples {DriverName, InfoList}, where InfoList is the result of calling info/1 for that DriverName. Only dynamically linked-in drivers are included in the list.

info(Name) -> InfoList
Types:
  Name = driver()
  InfoList = [InfoItem, ...]
  InfoItem = {Tag :: atom(), Value :: term()}

Returns a list of tuples {Tag, Value}, where Tag is the information item and Value is the result of calling info/2 with this driver name and this tag. The result is a tuple list containing all information available about a driver.

The following tags appears in the list:

- processes
- driver_options
- port_count
- linked_in_driver
- permanent
- awaiting_load
- awaiting_unload

For a detailed description of each value, see info/2.

The function throws a badarg exception if the driver is not present in the system.

info(Name, Tag) -> Value
Types:
  Name = driver()
  Tag = processes | driver_options | port_count | linked_in_driver |
permanent | awaiting_load | awaiting_unload
Value = term()

Returns specific information about one aspect of a driver. Parameter Tag specifies which aspect to get information about. The return Value differs between different tags:

processes
Returns all processes containing users of the specific drivers as a list of tuples \( (\text{pid}(), \text{integer()} \geq 0) \), where \( \text{integer()} \) denotes the number of users in process \( \text{pid}() \).

driver_options
Returns a list of the driver options provided when loading, and any options set by the driver during initialization. The only valid option is \text{kill_ports}.

port_count
Returns the number of ports (an \text{integer()} \geq 0) using the driver.

linked_in_driver
Returns a boolean(), which is true if the driver is a statically linked-in one, otherwise false.

permanent
Returns a boolean(), which is true if the driver has made itself permanent (and is not a statically linked-in driver), otherwise false.

awaiting_load
Returns a list of all processes having monitors for loading active. Each process is returned as \( (\text{pid}(), \text{integer()} \geq 0) \), where \( \text{integer()} \) is the number of monitors held by process \( \text{pid}() \).

awaiting_unload
Returns a list of all processes having monitors for unloading active. Each process is returned as \( (\text{pid}(), \text{integer()} \geq 0) \), where \( \text{integer()} \) is the number of monitors held by process \( \text{pid}() \).

If option \text{linked_in_driver} or \text{permanent} returns true, all other options return \text{linked_in_driver} or \text{permanent}, respectively.

The function throws a \text{badarg} exception if the driver is not present in the system or if the tag is not supported.

load(Path, Name) -> ok | {error, ErrorDesc}
Types:
Path = path()
Name = driver()
ErrorDesc = term()

Loads and links the dynamic driver \text{Name}. \text{Path} is a file path to the directory containing the driver. \text{Name} must be a sharable object/dynamic library. Two drivers with different \text{Path} parameters cannot be loaded under the same name. \text{Name} is a string or atom containing at least one character.

The \text{Name} specified is to correspond to the filename of the dynamically loadable object file residing in the directory specified as \text{Path}, but without the extension (that is, .so). The driver name provided in the driver initialization routine must correspond with the filename, in much the same way as Erlang module names correspond to the names of the .beam files.

If the driver was previously unloaded, but is still present because of open ports to it, a call to \text{load/2} stops the unloading and keeps the driver (as long as \text{Path} is the same), and \text{ok} is returned. If you really want the object code to
be reloaded, use \texttt{reload/2} or the low-level interface \texttt{try_load/3} instead. See also the description of different scenarios for loading/unloading in the introduction.

If more than one process tries to load an already loaded driver with the same \texttt{Path}, or if the same process tries to load it many times, the function returns \texttt{ok}. The emulator keeps track of the \texttt{load/2} calls, so that a corresponding number of \texttt{unload/2} calls must be done from the same process before the driver gets unloaded. It is therefore safe for an application to load a driver that is shared between processes or applications when needed. It can safely be unloaded without causing trouble for other parts of the system.

It is not allowed to load multiple drivers with the same name but with different \texttt{Path} parameters.

\begin{quote}
\textbf{Note:}
\end{quote}

\texttt{Path} is interpreted literally, so that all loaders of the same driver must specify the same \texttt{literal Path} string, although different paths can point out the same directory in the file system (because of use of relative paths and links).

On success, the function returns \texttt{ok}. On failure, the return value is \{\texttt{error,ErrorDesc}\}, where \texttt{ErrorDesc} is an opaque term to be translated into human readable form by function \texttt{format_error/1}.

For more control over the error handling, use the \texttt{try_load/3} interface instead.

The function throws a \texttt{badarg} exception if the parameters are not specified as described here.

\begin{verbatim}
load_driver(Path, Name) -> ok | {error, ErrorDesc}
\end{verbatim}

Types:
\begin{verbatim}
Path = path()
Name = driver()
ErrorDesc = term()
\end{verbatim}

Works essentially as \texttt{load/2}, but loads the driver with other options. All ports using the driver are killed with reason \texttt{driver_unloaded} when the driver is to be unloaded.

The number of loads and unloads by different users influences the loading and unloading of a driver file. The port killing therefore only occurs when the last user unloads the driver, or when the last process having loaded the driver exits.

This interface (or at least the name of the functions) is kept for backward compatibility. Using \texttt{try_load/3} with \{(\texttt{driver_options},[\texttt{kill_ports}])\} in the option list gives the same effect regarding the port killing.

The function throws a \texttt{badarg} exception if the parameters are not specified as described here.

\begin{verbatim}
loaded_drivers() -> {ok, Drivers}
\end{verbatim}

Types:
\begin{verbatim}
Drivers = [Driver]
Driver = string()
\end{verbatim}

Returns a list of all the available drivers, both (statically) linked-in and dynamically loaded ones.

The driver names are returned as a list of strings rather than a list of atoms for historical reasons.

For more information about drivers, see \texttt{info}.

\begin{verbatim}
monitor(Tag, Item) -> MonitorRef
\end{verbatim}

Types:
Tag = driver
Item = {Name, When}
Name = driver()
When = loaded | unloaded | unloaded_only
MonitorRef = reference()

Creates a driver monitor and works in many ways as erlang:monitor/2 in ERTS, does for processes. When a driver changes state, the monitor results in a monitor message that is sent to the calling process. MonitorRef returned by this function is included in the message sent.

As with process monitors, each driver monitor set only generates \textbf{one single message}. The monitor is “destroyed” after the message is sent, so it is then not needed to call demonitor/1.

MonitorRef can also be used in subsequent calls to demonitor/1 to remove a monitor.

The function accepts the following parameters:

\textbf{Tag}

The monitor tag is always \texttt{driver}, as this function can only be used to create driver monitors. In the future, driver monitors will be integrated with process monitors, why this parameter has to be specified for consistence.

\textbf{Item}

Parameter Item specifies which driver to monitor (the driver name) and which state change to monitor. The parameter is a tuple of arity two whose first element is the driver name and second element is one of the following:

\textbf{loaded}

Notifies when the driver is reloaded (or loaded if loading is underway). It only makes sense to monitor drivers that are in the process of being loaded or reloaded. A future driver name for loading cannot be monitored. That only results in a \texttt{DOWN} message sent immediately. Monitoring for loading is therefore most useful when triggered by function \texttt{try_load/3}, where the monitor is created \textbf{because} the driver is in such a pending state.

Setting a driver monitor for \texttt{loading} eventually leads to one of the following messages being sent:

\{'UP', \texttt{reference()}, \texttt{driver}, Name, loaded\}

This message is sent either immediately if the driver is already loaded and no reloading is pending, or when reloading is executed if reloading is pending.

The user is expected to know if reloading is demanded before creating a monitor for loading.

\{'UP', \texttt{reference()}, \texttt{driver}, Name, permanent\}

This message is sent if reloading was expected, but the (old) driver made itself permanent before reloading. It is also sent if the driver was permanent or statically linked-in when trying to create the monitor.

\{'DOWN', \texttt{reference()}, \texttt{driver}, Name, load_cancelled\}

This message arrives if reloading was underway, but the requesting user cancelled it by dying or calling \texttt{try_unload/2} (or \texttt{unload/1/unload_driver/1}) again before it was reloaded.

\{'DOWN', \texttt{reference()}, \texttt{driver}, Name, \{load_failure, Failure\}\}

This message arrives if reloading was underway but the loading for some reason failed. The Failure term is one of the errors that can be returned from \texttt{try_load/3}. The error term can be passed to \texttt{format_error/1} for translation into human readable form. Notice that the translation must be done in the same running Erlang virtual machine as the error was detected in.
unloaded

Monitors when a driver gets unloaded. If one monitors a driver that is not present in the system, one immediately gets notified that the driver got unloaded. There is no guarantee that the driver was ever loaded.

A driver monitor for unload eventually results in one of the following messages being sent:

{'DOWN', reference(), driver, Name, unloaded}

The monitored driver instance is now unloaded. As the unload can be a result of a reload/2 request, the driver can once again have been loaded when this message arrives.

{'UP', reference(), driver, Name, unload_cancelled}

This message is sent if unloading was expected, but while the driver was waiting for all ports to get closed, a new user of the driver appeared, and the unloading was cancelled.

This message appears if {ok, pending_driver} was returned from try_unload/2 for the last user of the driver, and then {ok, already_loaded} is returned from a call to try_load/3.

If one really wants to monitor when the driver gets unloaded, this message distorts the picture, because no unloading was done. Option unloaded_only creates a monitor similar to an unloaded monitor, but never results in this message.

{'UP', reference(), driver, Name, permanent}

This message is sent if unloading was expected, but the driver made itself permanent before unloading. It is also sent if trying to monitor a permanent or statically linked-in driver.

unloaded_only

A monitor created as unloaded_only behaves exactly as one created as unloaded except that the {'UP', reference(), driver, Name, unload_cancelled} message is never sent, but the monitor instead persists until the driver really gets unloaded.

The function throws a badarg exception if the parameters are not specified as described here.

reload(Path, Name) -> ok | {error, ErrorDesc}

Types:
Path = path()
Name = driver()
ErrorDesc = pending_process | OpaqueError
OpaqueError = term()

Reloads the driver named Name from a possibly different Path than previously used. This function is used in the code change scenario described in the introduction.

If there are other users of this driver, the function returns {error, pending_process}, but if there are no other users, the function call hangs until all open ports are closed.

Note:

Avoid mixing multiple users with driver reload requests.

To avoid hanging on open ports, use function try_load/3 instead.

The Name and Path parameters have exactly the same meaning as when calling the plain function load/2.

On success, the function returns ok. On failure, the function returns an opaque error, except the pending_process error described earlier. The opaque errors are to be translated into human readable form by function format_error/1.
For more control over the error handling, use the \texttt{try\_load/3} interface instead. The function throws a \texttt{badarg} exception if the parameters are not specified as described here.

\texttt{reload\_driver(Path, Name) \rightarrow ok \mid \{error, ErrorDesc\}}

\textbf{Types:}
- \texttt{Path = path()}
- \texttt{Name = driver()}
- \texttt{ErrorDesc = pending\_process | OpaqueError}
- \texttt{OpaqueError = term()}

Works exactly as \texttt{reload/2}, but for drivers loaded with the \texttt{load\_driver/2} interface. As this interface implies that ports are killed when the last user disappears, the function does not hang waiting for ports to get closed.

For more details, see \texttt{scenarios in this module description and the function description for reload/2.}

The function throws a \texttt{badarg} exception if the parameters are not specified as described here.

\texttt{try\_load(Path, Name, OptionList) \rightarrow}
- \{\texttt{ok, Status}\}
- \{\texttt{ok, PendingStatus, Ref}\}
- \{\texttt{error, ErrorDesc}\}

\textbf{Types:}
- \texttt{Path = path()}
- \texttt{Name = driver()}
- \texttt{OptionList = [Option]}
- \texttt{Option =}
  - \{\texttt{driver\_options, DriverOptionList}\}
  - \{\texttt{monitor, MonitorOption}\}
  - \{\texttt{reload, ReloadOption}\}
- \texttt{DriverOptionList = [DriverOption]}
- \texttt{DriverOption = kill\_ports}
- \texttt{MonitorOption = ReloadOption = pending\_driver | pending}
- \texttt{Status = loaded | already\_loaded | PendingStatus}
- \texttt{PendingStatus = pending\_driver | pending\_process}
- \texttt{Ref = reference()}
- \texttt{ErrorDesc = ErrorAtom | OpaqueError}
- \texttt{ErrorAtom =}
  - \texttt{linked\_in\_driver | inconsistent | permanent | not\_loaded\_by\_this\_process | not\_loaded | pending\_reload | pending\_process}
- \texttt{OpaqueError = term()}

Provides more control than the \texttt{load/2/reload/2} and \texttt{load\_driver/2/reload\_driver/2} interfaces. It never waits for completion of other operations related to the driver, but immediately returns the status of the driver as one of the following:

\{\texttt{ok, loaded}\}

The driver was loaded and is immediately usable.
{ok, already_loaded}

The driver was already loaded by another process or is in use by a living port, or both. The load by you is registered and a corresponding try_unload is expected sometime in the future.

{ok, pending_driver} or (ok, pending_driver, reference())

The load request is registered, but the loading is delayed because an earlier instance of the driver is still waiting to get unloaded (open ports use it). Still, unload is expected when you are done with the driver. This return value mostly occurs when options \{reload, pending_driver\} or \{reload, pending\} are used, but can occur when another user is unloading a driver in parallel and driver option kill_ports is set. In other words, this return value always needs to be handled.

{ok, pending_process} or (ok, pending_process, reference())

The load request is registered, but the loading is delayed because an earlier instance of the driver is still waiting to get unloaded by another user (not only by a port, in which case \{ok, pending_driver\} would have been returned). Still, unload is expected when you are done with the driver. This return value only occurs when option \{reload, pending\} is used.

When the function returns \{ok, pending_driver\} or \{ok, pending_process\}, one can get information about when the driver is actually loaded by using option \{monitor, MonitorOption\}.

When monitoring is requested, and a corresponding \{ok, pending_driver\} or \{ok, pending_process\} would be returned, the function instead returns a tuple \{ok, PendingStatus, reference()\} and the process then gets a monitor message later, when the driver gets loaded. The monitor message to expect is described in the function description of monitor/2.

**Note:**

In case of loading, monitoring can not only get triggered by using option \{reload, ReloadOption\}, but also in special cases where the load error is transient. Thus, \{monitor, pending_driver\} is to be used under basically all real world circumstances.

The function accepts the following parameters:

**Path**

The file system path to the directory where the driver object file is located. The filename of the object file (minus extension) must correspond to the driver name (used in parameter \Name\) and the driver must identify itself with the same name. Path can be provided as an iolist(), meaning it can be a list of other iolist()s, characters (8-bit integers), or binaries, all to be flattened into a sequence of characters.

The (possibly flattened) Path parameter must be consistent throughout the system. A driver is to, by all users, be loaded using the same literal Path. The exception is when reloading is requested, in which case Path can be specified differently. Notice that all users trying to load the driver later need to use the new Path if Path is changed using a reload option. This is yet another reason to have only one loader of a driver one wants to upgrade in a running system.

**Name**

This parameter is the name of the driver to be used in subsequent calls to function erlang:open_port in ERTS. The name can be specified as an iolist() or an atom(). The name specified when loading is used to find the object file (with the help of Path and the system-implied extension suffix, that is, .so). The name by which the driver identifies itself must also be consistent with this Name parameter, much as the module name of a Beam file much corresponds to its filename.
OptionList

Some options can be specified to control the loading operation. The options are specified as a list of two-tuples. The tuples have the following values and meanings:

\{driver_options, DriverOptionList\}

This is to provide options that changes its general behavior and "sticks" to the driver throughout its lifespan.

The driver options for a specified driver name need always to be consistent, even when the driver is reloaded, meaning that they are as much a part of the driver as the name.

The only allowed driver option is kill_ports, which means that all ports opened to the driver are killed with exit reason driver_unloaded when no process any longer has the driver loaded. This situation arises either when the last user calls try_unload/2, or when the last process having loaded the driver exits.

\{monitor, MonitorOption\}

A MonitorOption tells try_load/3 to trigger a driver monitor under certain conditions. When the monitor is triggered, the function returns a three-tuple \{ok, PendingStatus, reference()\}, where reference() is the monitor reference for the driver monitor.

Only one MonitorOption can be specified. It is one of the following:

- The atom pending, which means that a monitor is to be created whenever a load operation is delayed,
- The atom pending_driver, in which a monitor is created whenever the operation is delayed because of open ports to an otherwise unused driver.

Option pending_driver is of little use, but is present for completeness, as it is well defined which reload options that can give rise to which delays. However, it can be a good idea to use the same MonitorOption as the ReloadOption, if present.

If reloading is not requested, it can still be useful to specify option monitor, as forced unloads (driver option kill_ports or option kill_ports to try_unload/2) trigger a transient state where driver loading cannot be performed until all closing ports are closed. Thus, as try_unload can, in almost all situations, return \{ok, pending_driver\}, always specify at least \{monitor, pending_driver\} in production code (see the monitor discussion earlier).

\{reload, ReloadOption\}

This option is used to reload a driver from disk, most often in a code upgrade scenario. Having a reload option also implies that parameter Path does not need to be consistent with earlier loads of the driver.

To reload a driver, the process must have loaded the driver before, that is, there must be an active user of the driver in the process.

The reload option can be either of the following:

pending

With the atom pending, reloading is requested for any driver and is effectuated when all ports opened to the driver are closed. The driver replacement in this case takes place regardless if there are still pending users having the driver loaded.

The option also triggers port-killing (if driver option kill_ports is used) although there are pending users, making it usable for forced driver replacement, but laying much responsibility on the driver users. The pending option is seldom used as one does not want other users to have loaded the driver when code change is underway.
pending_driver

This option is more useful. Here, reloading is queued if the driver is **not** loaded by any other users, but the driver has opened ports, in which case \{ok, pending_driver\} is returned (a monitor option is recommended).

If the driver is unloaded (not present in the system), error code not_loaded is returned. Option reload is intended for when the user has already loaded the driver in advance.

The function can return numerous errors, some can only be returned given a certain combination of options. Some errors are opaque and can only be interpreted by passing them to function format_error/1, but some can be interpreted directly:

\{error,linked_in_driver\}

The driver with the specified name is an Erlang statically linked-in driver, which cannot be manipulated with this API.

\{error,inconsistent\}

The driver is already loaded with other DriverOptionList or a different literal Path argument. This can occur even if a reload option is specified, if DriverOptionList differs from the current.

\{error, permanent\}

The driver has requested itself to be permanent, making it behave like an Erlang linked-in driver and can no longer be manipulated with this API.

\{error, pending_process\}

The driver is loaded by other users when option \{reload, pending_driver\} was specified.

\{error, pending_reload\}

Driver reload is already requested by another user when option \{reload, ReloadOption\} was specified.

\{error, not_loaded_by_this_process\}

Appears when option reload is specified. The driver Name is present in the system, but there is no user of it in this process.

\{error, not_loaded\}

Appears when option reload is specified. The driver Name is not in the system. Only drivers loaded by this process can be reloaded.

All other error codes are to be translated by function format_error/1. Notice that calls to format_error are to be performed from the same running instance of the Erlang virtual machine as the error is detected in, because of system-dependent behavior concerning error values.

If the arguments or options are malformed, the function throws a badarg exception.

```erlang
try_unload(Name, OptionList) ->
    \{ok, Status\} |
    \{ok, PendingStatus, Ref\} |
    \{error, ErrorAtom\}
```

Types:
Name = driver()
OptionList = [Option]
Option = \{monitor, MonitorOption\} | kill_ports
MonitorOption = pending_driver | pending
Status = unloaded | PendingStatus
PendingStatus = pending_driver | pending_process
Ref = reference()
ErrorAtom = [linked_in_driver | not_loaded | not_loaded_by_this_process | permanent]

This is the low-level function to unload (or decrement reference counts of) a driver. It can be used to force port killing, in much the same way as the driver option kill_ports implicitly does. Also, it can trigger a monitor either because other users still have the driver loaded or because open ports use the driver.

Unloading can be described as the process of telling the emulator that this particular part of the code in this particular process (that is, this user) no longer needs the driver. That can, if there are no other users, trigger unloading of the driver, in which case the driver name disappears from the system and (if possible) the memory occupied by the driver executable code is reclaimed.

If the driver has option kill_ports set, or if kill_ports is specified as an option to this function, all pending ports using this driver are killed when unloading is done by the last user. If no port-killing is involved and there are open ports, the unloading is delayed until no more open ports use the driver. If, in this case, another user (or even this user) loads the driver again before the driver is unloaded, the unloading never takes place.

To allow the user to request unloading to wait for actual unloading, monitor triggers can be specified in much the same way as when loading. However, as users of this function seldom are interested in more than decrementing the reference counts, monitoring is seldom needed.

**Note:**

If option kill_ports is used, monitor triggering is crucial, as the ports are not guaranteed to be killed until the driver is unloaded. Thus, a monitor must be triggered for at least the pending_driver case.

The possible monitor messages to expect are the same as when using option unloaded to function monitor/2.

The function returns one of the following statuses upon success:

\{ok, unloaded\}

The driver was immediately unloaded, meaning that the driver name is now free to use by other drivers and, if the underlying OS permits it, the memory occupied by the driver object code is now reclaimed.

The driver can only be unloaded when there are no open ports using it and no more users require it to be loaded.

\{ok, pending_driver\} or \{ok, pending_driver, reference()\}

Indicates that this call removed the last user from the driver, but there are still open ports using it. When all ports are closed and no new users have arrived, the driver is reloaded and the name and memory reclaimed.

This return value is valid even if option kill_ports was used, as killing ports can be a process that does not complete immediately. However, the condition is in that case transient. Monitors are always useful to detect when the driver is really unloaded.

\{ok, pending_process\} or \{ok, pending_process, reference()\}

The unload request is registered, but other users still hold the driver. Notice that the term pending_process can refer to the running process; there can be more than one user in the same process.
This is a normal, healthy, return value if the call was just placed to inform the emulator that you have no further use of the driver. It is the most common return value in the most common scenario described in the introduction.

The function accepts the following parameters:

Name

Name is the name of the driver to be unloaded. The name can be specified as an iolist() or as an atom().

OptionList

Argument OptionList can be used to specify certain behavior regarding ports and triggering monitors under certain conditions:

kill_ports

Forces killing of all ports opened using this driver, with exit reason driver_unloaded, if you are the last user of the driver.

If other users have the driver loaded, this option has no effect.

To get the consistent behavior of killing ports when the last user unloads, use driver option kill_ports when loading the driver instead.

{monitor, MonitorOption}

Creates a driver monitor if the condition specified in MonitorOption is true. The valid options are:

pending_driver

Creates a driver monitor if the return value is to be {ok, pending_driver}.

pending

Creates a monitor if the return value is {ok, pending_driver} or {ok, pending_process}.

The pending_driver MonitorOption is by far the most useful. It must be used to ensure that the driver really is unloaded and the ports closed whenever option kill_ports is used, or the driver can have been loaded with driver option kill_ports.

Using the monitor triggers in the call to try_unload ensures that the monitor is added before the unloading is executed, meaning that the monitor is always properly triggered, which is not the case if monitor/2 is called separately.

The function can return the following error conditions, all well specified (no opaque values):

{error, linked_in_driver}

You were trying to unload an Erlang statically linked-in driver, which cannot be manipulated with this interface (and cannot be unloaded at all).

{error, not_loaded}

The driver Name is not present in the system.

{error, not_loaded_by_this_process}

The driver Name is present in the system, but there is no user of it in this process.

As a special case, drivers can be unloaded from processes that have done no corresponding call to try_load/3 if, and only if, there are no users of the driver at all, which can occur if the process containing the last user dies.

{error, permanent}

The driver has made itself permanent, in which case it can no longer be manipulated by this interface (much like a statically linked-in driver).

The function throws a badarg exception if the parameters are not specified as described here.
unload(Name) -> ok | {error, ErrorDesc}
Types:
   Name = driver()
   ErrorDesc = term()
Unloads, or at least dereferences the driver named Name. If the caller is the last user of the driver, and no more open ports use the driver, the driver gets unloaded. Otherwise, unloading is delayed until all ports are closed and no users remain.
If there are other users of the driver, the reference counts of the driver is merely decreased, so that the caller is no longer considered a user of the driver. For use scenarios, see the description in the beginning of this module.
The ErrorDesc returned is an opaque value to be passed further on to function format_error/1. For more control over the operation, use the try_unload/2 interface.
The function throws a badarg exception if the parameters are not specified as described here.

unload_driver(Name) -> ok | {error, ErrorDesc}
Types:
   Name = driver()
   ErrorDesc = term()
Unloads, or at least dereferences the driver named Name. If the caller is the last user of the driver, all remaining open ports using the driver are killed with reason driver_unloaded and the driver eventually gets unloaded.
If there are other users of the driver, the reference counts of the driver is merely decreased, so that the caller is no longer considered a user. For use scenarios, see the description in the beginning of this module.
The ErrorDesc returned is an opaque value to be passed further on to function format_error/1. For more control over the operation, use the try_unload/2 interface.
The function throws a badarg exception if the parameters are not specified as described here.

See Also
erts:erl_driver(4), erts:driver_entry(4)
erl_epmd

Erlang module

This module communicates with the EPMD daemon, see epmd. To implement your own epmd module please see ERTS User's Guide: How to Implement an Alternative Node Discovery for Erlang Distribution

Exports

start_link() -> {ok, pid()} | ignore | {error, term()}
This function is invoked as this module is added as a child of the erl_distribution supervisor.

register_node(Name, Port) -> Result
register_node(Name, Port, Driver) -> Result
Types:
  Name = string()
  Port = integer() >= 0
  Driver = inet_tcp | inet6_tcp | inet | inet6
  Creation = integer() >= 0 | -1
  Result = {ok, Creation} | {error, already_registered} | term()
 Registers the node with epmd and tells epmd what port will be used for the current node. It returns a creation number. This number is incremented on each register to help with identifying if a node is reconnecting to epmd.

port_please(Name, Host) -> {ok, Port, Version} | noport
port_please(Name, Host, Timeout) -> {port, Port, Version} | noport
Types:
  Name = atom() | string()
  Host = atom() | string() | inet:ip_address()
  Timeout = integer() >= 0 | infinity
  Port = Version = integer() >= 0
Requests the distribution port for the given node of an EPMD instance. Together with the port it returns a distribution protocol version which has been 5 since Erlang/OTP R6.

listen_port_please(Name, Host) -> {ok, Port}
Types:
  Name = atom()
  Host = string() | inet:ip_address()
  Port = integer() >= 0
Called by the distribution module to get which port the local node should listen to when accepting new distribution requests.

address_please(Name, Host, AddressFamily) ->
  Success | {error, term()}
Types:
erl_epmd

Name = string()
Host = string() | inet:ip_address()
AddressFamily = inet | inet6
Port = Version = integer() >= 0
Success =
  {ok, inet:ip_address()} |
  {ok, inet:ip_address(), Port, Version}

Called by the distribution module to resolves the Host to an IP address of a remote node.
As an optimization this function may also return the port and version of the remote node. If port and version are returned port_please/3 will not be called.

names(Host) -> {ok, [{Name, Port}]} | {error, Reason}

Types:
  Host = atom() | string() | inet:ip_address()
  Name = string()
  Port = integer() >= 0
  Reason = address | file:posix()

Called by net_adm:names/0. Host defaults to the localhost. Returns the names and associated port numbers of the Erlang nodes that epmd registered at the specified host. Returns {error, address} if epmd is not operational.

Example:

(arnes@dunn)1> erl_epmd:names(localhost).
{ok,[{"arnes",40262}]}
erl_prim_loader

Erlang module

The module erl_prim_loader is moved to the runtime system application. Please see erl_prim_loader(3) in the ERTS reference manual instead.
erlang

Erlang module

The module erlang is moved to the runtime system application. Please see erlang(3) in the ERTS reference manual instead.
erpc

Erlang module

This module provides services similar to Remote Procedure Calls (RPC). A remote procedure call is a method to call a function on a remote node and collect the answer. It is used for collecting information on a remote node, or for running a function with some specific side effects on the remote node.

This is an enhanced subset of the operations provided by the rpc module. Enhanced in the sense that it makes it possible to distinguish between returned value, raised exceptions, and other errors. erpc also has better performance and scalability than the original rpc implementation. However, current rpc module will utilize erpc in order to also provide these properties when possible.

In order for an erpc operation to succeed, the remote node also needs to support erpc. Typically only ordinary Erlang nodes as of OTP 23 have erpc support.

Note that it is up to the user to ensure that correct code to execute via erpc is available on the involved nodes.

Data Types

request_id()

An opaque type of call request identifiers. For more information see send_request/4.

Exports

call(Node, Fun) -> Result

call(Node, Fun, Timeout) -> Result

Types:

Node = node()
Fun = function()
Timeout = 0..4294967295 | infinity
Result = term()

The same as calling erpc:call(Node,erlang,apply,[Fun,[]],Timeout). May raise all the same exceptions as erpc:call/5 plus an {erpc, badarg} error exception if Fun is not a fun of zero arity.

The call erpc:call(Node,Fun) is the same as the call erpc:call(Node,Fun,infinity).

call(Node, Module, Function, Args) -> Result

call(Node, Module, Function, Args, Timeout) -> Result

Types:

Node = node()
Module = Function = atom()
Args = [term()]
Timeout = 0..4294967295 | infinity
Result = term()

Evaluates apply(Module, Function, Args) on node Node and returns the corresponding value Result. Timeout is an integer representing the timeout in milliseconds or the atom infinity which prevents the operation from ever timing out.
The call `erpc:call(Node, Module, Function, Args)` is equivalent to the call `erpc:call(Node, Module, Function, Args, infinity)`.

The `call()` function only returns if the applied function successfully returned without raising any uncaught exceptions, the operation did not time out, and no failures occurred. In all other cases an exception is raised. The following exceptions, listed by exception class, can currently be raised by `erpc:call()`:

**throw**

The applied function called `throw(Value)` and did not catch this exception. The exception reason `Value` equals the argument passed to `throw/1.`

**exit**

Exception reason:

*{exception, ExitReason}*  
The applied function called `exit(ExitReason)` and did not catch this exception. The exit reason `ExitReason` equals the argument passed to `exit/1.`

*{signal, ExitReason}*  
The process that applied the function received an exit signal and terminated due to this signal. The process terminated with exit reason `ExitReason`.

**error**

Exception reason:

*{exception, ErrorReason, StackTrace}*  
A runtime error occurred which raised and error exception while applying the function, and the applied function did not catch the exception. The error reason `ErrorReason` indicates the type of error that occurred. `StackTrace` is formatted as when caught in a `try/catch` construct. The `StackTrace` is limited to the applied function and functions called by it.

*{erpc, ERpcErrorReason}*  
The erpc operation failed. The following ERpcErrorReasons are the most common ones:

**badarg**

If any one of these are true:

- Node is not an atom.
- Module is not an atom.
- Function is not an atom.
- Args is not a list. Note that the list is not verified to be a proper list at the client side.
- Timeout is not the atom `infinity` or an integer in valid range.

**noconnection**

The connection to Node was lost or could not be established. The function may or may not be applied.

**system_limit**

The erpc operation failed due to some system limit being reached. This typically due to failure to create a process on the remote node Node, but can be other things as well.

**timeout**

The erpc operation timed out. The function may or may not be applied.
The remote node Node does not support this erpc operation.

If the erpc operation fails, but it is unknown if the function is/will be applied (that is, a timeout or a connection loss), the caller will not receive any further information about the result if/when the applied function completes. If the applied function explicitly communicates with the calling process, such communication may, of course, reach the calling process.

**Note:**
You cannot make any assumptions about the process that will perform the apply(). It may be the calling process itself, a server, or a freshly spawned process.

cast(Node, Fun) -> ok
Types:
- Node = node()
- Fun = function()

The same as calling erpc:cast(Node,erlang,apply,[Fun,[]]).

erpc:cast/2 fails with an {erpc, badarg} error exception if:
- Node is not an atom.
- Fun is not a fun of zero arity.

cast(Node, Module, Function, Args) -> ok
Types:
- Node = node()
- Module = Function = atom()
- Args = [term()]

Evaluates apply(Module, Function, Args) on node Node. No response is delivered to the calling process.
erpc:cast() returns immediately after the cast request has been sent. Any failures beside bad arguments are silently ignored.

erpc:cast/4 fails with an {erpc, badarg} error exception if:
- Node is not an atom.
- Module is not an atom.
- Function is not an atom.
- Args is not a list. Note that the list is not verified to be a proper list at the client side.

**Note:**
You cannot make any assumptions about the process that will perform the apply(). It may be a server, or a freshly spawned process.

check_response(Message, RequestId) ->
{response, Result} | no_response
Types:
```
Message = term()
RequestId = request_id()
Result = term()
```

Check if a message is a response to a call request previously made by the calling process using `erpc:send_request/4`. `RequestId` should be the value returned from the previously made `erpc:send_request()` call, and the corresponding response should not already have been received and handled to completion by `erpc:check_response()`, `erpc:receive_response()`, or `erpc:wait_response()`. `Message` is the message to check.

If `Message` does not correspond to the response, the atom `no_response` is returned. If `Message` corresponds to the response, the call operation is completed and either the result is returned as `{response, Result}` where `Result` corresponds to the value returned from the applied function or an exception is raised. The exceptions that can be raised corresponds to the same exceptions as can be raised by `erpc:call/4`. That is, no `{erpc, timeout}` error exception can be raised. `erpc:check_response()` will fail with an `{erpc, badarg}` exception if/when an invalid `RequestId` is detected.

If the `erpc` operation fails, but it is unknown if the function is/will be applied (that is, a connection loss), the caller will not receive any further information about the result if/when the applied function completes. If the applied function explicitly communicates with the calling process, such communication may, of course, reach the calling process.

```
multicall(Nodes, Fun) -> Result
multicall(Nodes, Fun, Timeout) -> Result
```

Types:
- `Nodes` = [atom()]
- `Fun` = function()
- `Timeout` = 0..4294967295 | infinity
- `Result` = term()

The same as calling `erpc:multicall(Nodes,erlang,apply,[Fun,[]],Timeout)`. May raise all the same exceptions as `erpc:multicall/5` plus an `{erpc, badarg}` error exception if `Fun` is not a fun of zero arity.

The call `erpc:multicall(Nodes,Fun)` is the same as the call `erpc:multicall(Nodes,Fun, infinity)`.

```
multicall(Nodes, Module, Function, Args) -> Result
multicall(Nodes, Module, Function, Args, Timeout) -> Result
```

Types:
- `Nodes` = [atom()]
- `Module` = Function = atom()
- `Args` = [term()]
- `Timeout` = 0..4294967295 | infinity
- `Result` = [{ok, ReturnValue :: term()} | caught_call_exception()]

\[
\text{caught_call_exception()} =
\{\text{throw, Throw :: term()}} \mid
\{\text{exit, \{exception, Reason :: term()}}\mid
\{\text{error},
\{\text{exception, Reason :: term()}, \text{StackTrace :: [stack_item()]}\}\mid
\{\text{exit, \{signal, Reason :: term()}}\mid
\]

116 | Ericsson AB. All Rights Reserved.: Kernel
{error, {erpc, Reason :: term()}}

stack_item() =
  {Module :: atom(),
   Function :: atom(),
   Arity :: arity() | (Args :: [term()]),
   Location ::
     [{file, Filename :: string()} | {line, Line :: integer() >= 1}]}

Performs multiple call operations in parallel on multiple nodes. That is, evaluates apply(Module, Function, Args) on the nodes Nodes in parallel. Timeout is an integer representing the timeout in milliseconds or the atom infinity which prevents the operation from ever timing out. The result is returned as a list where the result from each node is placed at the same position as the node name is placed in Nodes. Each item in the resulting list is formatted as either:

{ok, Result}

The call operation for this specific node returned Result.

{Class, ExceptionReason}

The call operation for this specific node raised an exception of class Class with exception reason ExceptionReason. These corresponds the the exceptions that erpc:call/5 can raise.

erpc:multicall/5 fails with an {erpc, badarg} error exception if:

• Nodes is not a proper list of atoms. Note that some requests may already have been sent when the failure occurs. That is, the function may or may not be applied on some nodes.
• Module is not an atom.
• Function is not an atom.
• Args is not a list. Note that the list is not verified to be a proper list at the client side.

The call erpc:multicall(Nodes, Module, Function, Args) is equivalent to the call erpc:multicall(Nodes, Module, Function, Args, infinity). These calls are also equivalent to calling my_multicall(Nodes, Module, Function, Args) if one disregard performance and failure behavior:

```erlang
my_multicall(Nodes, Module, Function, Args) ->
  ReqIds = lists:map(fun (Node) ->
                       erpc:send_request(Node, Module, Function, Args)
                       end, Nodes),
  lists:map(fun (ReqId) ->
            try
              {ok, erpc:receive_response(ReqId, infinity)}
            catch
              Class:Reason ->
                {Class, Reason}
            end
            end, ReqIds).
```

The Timeout value in milliseconds sets an upper time limit for all call operations to complete.

If an erpc operation fails, but it is unknown if the function is/will be applied (that is, a timeout, connection loss, or an improper Nodes list), the caller will not receive any further information about the result if/when the applied function completes. If the applied function communicates with the calling process, such communication may, of course, reach the calling process.
You cannot make any assumptions about the process that will perform the apply(). It may be the calling process itself, a server, or a freshly spawned process.

multicast(Nodes, Fun) -> ok
Types:
  Nodes = [node()]
  Fun = function()
The same as calling erpc:multicast(Nodes,erlang,apply,[Fun,[[]]]).

erpc:multicast/2 fails with an {erpc, badarg} error exception if:
  • Nodes is not a proper list of atoms.
  • Fun is not a a fun of zero arity.

multicast(Nodes, Module, Function, Args) -> ok
Types:
  Nodes = [node()]
  Module = Function = atom()
  Args = [term()]
Evaluates apply(Module, Function, Args) on the nodes Nodes. No response is delivered to the calling process. erpc:multicast() returns immediately after the cast requests have been sent. Any failures beside bad arguments are silently ignored.

erpc:multicast/4 fails with an {erpc, badarg} error exception if:
  • Nodes is not a proper list of atoms. Note that some requests may already have been sent when the failure occurs. That is, the function may or may not be applied on some nodes.
  • Module is not an atom.
  • Function is not an atom.
  • Args is not a list. Note that the list is not verified to be a proper list at the client side.

You cannot make any assumptions about the process that will perform the apply(). It may be a server, or a freshly spawned process.

receive_response(RequestId) -> Result
receive_response(RequestId, Timeout) -> Result
Types:
  RequestId = request_id()
  Timeout = 0 .. 4294967295 | infinity
  Result = term()
Receive a response to a call request previously made by the calling process using erpc:send_request/4. RequestId should be the value returned from the previously made erpc:send_request() call, and the corresponding response should not already have been received and handled to completion by
erpc:check_response(), erpc:receive_response(), or erpc:wait_response(). Timeout is an integer representing the timeout in milliseconds or the atom infinity which prevents the operation from ever timing out. The call operation is completed once the erpc:receive_response() call returns or raise an exception.

The call erpc:receive_response(RequestId) is equivalent to the call erpc:receive_response(RequestId, infinity).

A call to the function my_call(Node, Module, Function, Args, Timeout) below is equivalent to the call erpc:call(Node, Module, Function, Args, Timeout) if one disregards performance. erpc:call() can utilize a message queue optimization which removes the need to scan the whole message queue which the combination erpc:send_request()/erpc:receive_response() cannot.

```
my_call(Node, Module, Function, Args, Timeout) ->
  RequestId = erpc:send_request(Node, Module, Function, Args),
  erpc:receive_response(RequestId, Timeout).
```

If the erpc operation fails, but it is unknown if the function is/will be applied (that is, a timeout, or a connection loss), the caller will not receive any further information about the result if/when the applied function completes. If the applied function explicitly communicates with the calling process, such communication may, of course, reach the calling process.

erpc:receive_response() will return or raise exceptions the same way as erpc:call/5 does with the exception of {erpc, badarg}. An {erpc, badarg} exception will be raised if/when an invalid RequestId is detected or if an invalid Timeout is passed.

send_request(Node, Fun) -> RequestId

Types:

Node = node()
Fun = function()
RequestId = request_id()

The same as calling erpc:send_request(Node,erlang,apply,[Fun,[]]).

erpc:send_request/2 fails with an (erpc, badarg) error exception if:

• Node is not an atom.
• Fun is not a fun of zero arity.

Note:

You cannot make any assumptions about the process that will perform the apply(). It may be a server, or a freshly spawned process.

send_request(Node, Module, Function, Args) -> RequestId

Types:
Node = node()
Module = Function = atom()
Args = [term()]
RequestId = request_id()

Send an asynchronous call request to the node Node. erpc:send_request() returns a request identifier that later is to be passed as argument to either erpc:receive_response(), erpc:wait_response(), or erpc:check_response() in order to get the response of the call request.

erpc:send_request() fails with an {erpc, badarg} error exception if:

- Node is not an atom.
- Module is not an atom.
- Function is not an atom.
- Args is not a list. Note that the list is not verified to be a proper list at the client side.

wait_response(RequestId) -> {response, Result} | no_response
wait_response(RequestId, WaitTime) ->
  {response, Result} | no_response

Types:
  RequestId = request_id()
  WaitTime = 0..4294967295 | infinity
  Result = term()

Wait or poll for a response message to a call request previously made by the calling process using erpc:send_request/4. RequestId should be the value returned from the previously made erpc:send_request() call, and the corresponding response should not already have been received and handled to completion by erpc:check_response(), erpc:receive_response(), or erpc:wait_response(). WaitTime equals the time to wait in milliseconds (or the atom infinity) during the wait. WaitTime is an integer representing time to wait in milliseconds or the atom infinity which will cause wait_response/2 to wait for a response until it appears regardless of how long time that is.

The call erpc:wait_response(RequestId) is equivalent to the call erpc:wait_response(RequestId, 0). That is, poll for a response message to a call request previously made by the calling process.

If no response is received before WaitTime milliseconds, the atom no_response is returned. It is valid to continue waiting for a response as many times as needed up until a response has been received and completed by erpc:check_response(), erpc:receive_response(), or erpc:wait_response(). If a response is received, the call operation is completed and either the result is returned as {response, Result} where Result corresponds to the value returned from the applied function or an exception is raised. The exceptions that can be raised corresponds to the same exceptions as can be raised by erpc:call/4. That is, no {erpc, timeout} error exception can be raised. erpc:wait_response() will fail with an {erpc, badarg} exception if/when an invalid RequestId is detected or if an invalid WaitTime is passed.

If the erpc operation fails, but it is unknown if the function is/will be applied (that is, a too large wait time value, or a connection loss), the caller will not receive any further information about the result if/when the applied function completes. If the applied function explicitly communicates with the calling process, such communication may, of course, reach the calling process.
error_handler

Erlang module

This module defines what happens when certain types of errors occur.

Exports

raise_undef_exception(Module, Function, Args) -> no_return()

Types:
  Module = Function = atom()
  Args = list()
  A (possibly empty) list of arguments Arg1,..,ArgN

Raises an undef exception with a stacktrace, indicating that Module:Function/N is undefined.

undefined_function(Module, Function, Args) -> any()

Types:
  Module = Function = atom()
  Args = list()
  A (possibly empty) list of arguments Arg1,..,ArgN

This function is called by the runtime system if a call is made to Module:Function(Arg1,.., ArgN) and Module:Function/N is undefined. Notice that this function is evaluated inside the process making the original call.

This function first attempts to autoload Module. If that is not possible, an undef exception is raised.

If it is possible to load Module and function Function/N is exported, it is called.

Otherwise, if function '$handle_undefined_function'/2 is exported, it is called as '$handle_undefined_function'(Function, Args).

Warning:

Defining '$handle_undefined_function'/2 in ordinary application code is highly discouraged. It is very easy to make subtle errors that can take a long time to debug. Furthermore, none of the tools for static code analysis (such as Dialyzer and Xref) supports the use of '$handle_undefined_function'/2 and no such support will be added. Only use this function after having carefully considered other, less dangerous, solutions. One example of potential legitimate use is creating stubs for other sub-systems during testing and debugging.

Otherwise an undef exception is raised.

undefined_lambda(Module, Fun, Args) -> term()

Types:
  Module = atom()
  Fun = function()
  Args = list()
  A (possibly empty) list of arguments Arg1,..,ArgN

This function is evaluated if a call is made to Fun(Arg1,.., ArgN) when the module defining the fun is not loaded. The function is evaluated inside the process making the original call.
If `Module` is interpreted, the interpreter is invoked and the return value of the interpreted `Fun(Arg1,.., ArgN)` call is returned.

Otherwise, it returns, if possible, the value of `apply(Fun, Args)` after an attempt is made to autoload `Module`. If this is not possible, the call fails with exit reason `undef`.

**Notes**

The code in `error_handler` is complex. Do not change it without fully understanding the interaction between the error handler, the `init` process of the code server, and the I/O mechanism of the code.

Code changes that seem small can cause a deadlock, as unforeseen consequences can occur. The use of `input` is dangerous in this type of code.
error_logger

Erlang module

Note:

In Erlang/OTP 21.0, a new API for logging was added. The old error_logger module can still be used by legacy code, but log events are redirected to the new Logger API. New code should use the Logger API directly.

error_logger is no longer started by default, but is automatically started when an event handler is added with error_logger:add_report_handler/1,2. The error_logger module is then also added as a handler to the new logger.

See logger(3) and the Logging chapter in the User's Guide for more information.

The Erlang error_logger is an event manager (see OTP Design Principles and gen_event(3)), registered as error_logger.

Error logger is no longer started by default, but is automatically started when an event handler is added with add_report_handler/1,2. The error_logger module is then also added as a handler to the new logger, causing log events to be forwarded from logger to error logger, and consequently to all installed error logger event handlers.

User-defined event handlers can be added to handle application-specific events.

Existing event handlers provided by STDLIB and SASL are still available, but are no longer used by OTP.

Warning events were introduced in Erlang/OTP R9C and are enabled by default as from Erlang/OTP 18.0. To retain backwards compatibility with existing user-defined event handlers, the warning events can be tagged as errors or info using command-line flag +W <e | i | w>, thus showing up as ERROR REPORT or INFO REPORT in the logs.

Data Types

report() =
   [{Tag :: term(), Data :: term()} | term()] | string() | term()

Exports

add_report_handler(Handler) -> any()
add_report_handler(Handler, Args) -> Result

Types:
   Handler = module()
   Args = gen_event:handler_args()
   Result = gen_event:add_handler_ret()

Adds a new event handler to the error logger. The event handler must be implemented as a gen_event callback module, see gen_event(3).

Handler is typically the name of the callback module and Args is an optional term (defaults to []) passed to the initialization callback function Handler:init/1. The function returns ok if successful.

The event handler must be able to handle the events in this module, see section Events.
The first time this function is called, `error_logger` is added as a Logger handler, and the `error_logger` process is started.

```erlang
delete_report_handler(Handler) -> Result
```

Types:
```
   Handler = module()
   Result = gen_event:del_handler_ret()
```

Deletes an event handler from the error logger by calling `gen_event:delete_handler(error_logger, Handler, [])`, see `gen_event(3)`.

If no more event handlers exist after the deletion, `error_logger` is removed as a Logger handler, and the `error_logger` process is stopped.

```erlang
error_msg(Format) -> ok
error_msg(Format, Data) -> ok
format(Format, Data) -> ok
```

Types:
```
   Format = string()
   Data = list()
```

Log a standard error event. The `Format` and `Data` arguments are the same as the arguments of `io:format/2` in STDLIB.

Error logger forwards the event to Logger, including metadata that allows backwards compatibility with legacy error logger event handlers.

The event is handled by the default Logger handler.

These functions are kept for backwards compatibility and must not be used by new code. Use the `?LOG_ERROR` macro or `logger:error/1,2,3` instead.

**Example:**

```erlang
1> error_logger:error_msg("An error occurred in ~p", [a_module]).
=ERROR REPORT==== 22-May-2018::11:18:43.376917 ===
An error occurred in a_module
ok
```

**Warning:**

If the Unicode translation modifier (t) is used in the format string, all event handlers must ensure that the formatted output is correctly encoded for the I/O device.

```erlang
error_report(Report) -> ok
```

Types:
```
   Report = report()
```

Log a standard error event. Error logger forwards the event to Logger, including metadata that allows backwards compatibility with legacy error logger event handlers.

The event is handled by the default Logger handler.
This function is kept for backwards compatibility and must not be used by new code. Use the \_\_LOG\_ERROR macro or \_\_logger:~error/1, 2, 3 instead.

**Example:**

```erlang
2> error_logger:error_report([{tag1, data1}, a_term, {tag2, data}]).
=ERROR REPORT==== 22-May-2018::11:24:23.699306 ===
tag1: data1
 a_term
 tag2: data
ok
3> error_logger:error_report("Serious error in my module").
=ERROR REPORT==== 22-May-2018::11:24:45.972445 ===
Serious error in my module
ok
```

**error_report**

**Types:**

- **Type** = term()
- **Report** = report()

Log a user-defined error event. Error logger forwards the event to Logger, including metadata that allows backwards compatibility with legacy error logger event handlers.

Error logger also adds a `domain` field with value `[Type]` to this event's metadata, causing the filters of the default Logger handler to discard the event. A different Logger handler, or an error logger event handler, must be added to handle this event.

It is recommended that `Report` follows the same structure as for `error_report/1`.

This function is kept for backwards compatibility and must not be used by new code. Use the \_\_LOG\_ERROR macro or \_\_logger:~error/1, 2, 3 instead.

**get_format_depth**

**Returns:** \_\_max(10, Depth), where `Depth` is the value of `error_logger_format_depth` in the Kernel application, if `Depth` is an integer. Otherwise, `unlimited` is returned.

**Note:**

The `error_logger_format_depth` variable is deprecated since the Logger API was introduced in Erlang/OTP 21.0. The variable, and this function, are kept for backwards compatibility since they still might be used by legacy report handlers.

**info_msg**

**Types:**

- **Format** = string()
- **Data** = list()

Log a standard information event. The `Format` and `Data` arguments are the same as the arguments of `io:format/2` in STDLIB.

Error logger forwards the event to Logger, including metadata that allows backwards compatibility with legacy error logger event handlers.
error_logger

The event is handled by the default Logger handler.
These functions are kept for backwards compatibility and must not be used by new code. Use the \?LOG_INFO macro or \logger:info/1,2,3 instead.

**Example:**

```
1> error_logger:info_msg("Something happened in ~p", [a_module]).
 ERROR REPORT==== 22-May-2018::12:03:32.612462 ====
 Something happened in a_module
 ok
```

**Warning:**
If the Unicode translation modifier (t) is used in the format string, all event handlers must ensure that the formatted output is correctly encoded for the I/O device.

### info_report(Report) -> ok

**Types:**
- **Report** = report()

Log a standard information event. Error logger forwards the event to Logger, including metadata that allows backwards compatibility with legacy error logger event handlers.
The event is handled by the default Logger handler.
This functions is kept for backwards compatibility and must not be used by new code. Use the \?LOG_INFO macro or \logger:info/1,2,3 instead.

**Example:**

```
2> error_logger:info_report([\{tag1, data1\}, a_term, \{tag2, data\}]).
 ERROR REPORT==== 22-May-2018::12:06:35.994440 ====
 tag1: data1
 a_term
 tag2: data
 ok
3> error_logger:info_report("Something strange happened").
 ERROR REPORT==== 22-May-2018::12:06:49.866872 ====
 Something strange happened
 ok
```

### info_report(Type, Report) -> ok

**Types:**
- **Type** = any()
- **Report** = report()

Log a user-defined information event. Error logger forwards the event to Logger, including metadata that allows backwards compatibility with legacy error logger event handlers.
Error logger also adds a domain field with value \[Type\] to this event's metadata, causing the filters of the default Logger handler to discard the event. A different Logger handler, or an error logger event handler, must be added to handle this event.
It is recommended that Report follows the same structure as for info_report/1.
This function is kept for backwards compatibility and must not be used by new code. Use the `?LOG_INFO` macro or `logger:info/1,2,3` instead.

```erlang
logfile(Request :: {open, Filename}) -> ok | {error, OpenReason}
logfile(Request :: close) -> ok | {error, CloseReason}
logfile(Request :: filename) -> Filename | {error, FilenameReason}
```

Types:
- `Filename = file:name()`
- `OpenReason = allready_have_logfile | open_error()`
- `CloseReason = module_not_found`
- `FilenameReason = no_log_file`
- `open_error() = file:posix() | badarg | system_limit`

Enables or disables printout of standard events to a file.

This is done by adding or deleting the `error_logger_file_handler` event handler, and thus indirectly adding `error_logger` as a Logger handler.

Notice that this function does not manipulate the Logger configuration directly, meaning that if the default Logger handler is already logging to a file, this function can potentially cause logging to a second file.

This function is useful as a shortcut during development and testing, but must not be used in a production system. See section Logging in the Kernel User's Guide, and the `logger(3)` manual page for information about how to configure Logger for live systems.

Request is one of the following:

- `{open, Filename}`
  - Opens log file `Filename`. Returns `ok` if successful, or `{error, allready_have_logfile}` if logging to file is already enabled, or an error tuple if another error occurred (for example, if `Filename` cannot be opened).
  - The file is opened with encoding UTF-8.

- `close`
  - Closes the current log file. Returns `ok`, or `{error, module_not_found}`.

- `filename`
  - Returns the name of the log file `Filename`, or `{error, no_log_file}` if logging to file is not enabled.

```erlang
tty(Flag) -> ok
```

Types:
- `Flag = boolean()`

Enables (`Flag == true`) or disables (`Flag == false`) printout of standard events to the terminal.

This is done by manipulating the Logger configuration. The function is useful as a shortcut during development and testing, but must not be used in a production system. See section Logging in the Kernel User's Guide, and the `logger(3)` manual page for information about how to configure Logger for live systems.

```erlang
warning_map() -> Tag
```

Types:
Tag = error | warning | info

Returns the current mapping for warning events. Events sent using warning_msg/1,2 or warning_report/1,2 are tagged as errors, warnings (default), or info, depending on the value of command-line flag +W.

Example:

```erl
os$ erl
Erlang (BEAM) emulator version 5.4.8 [hipe] [threads:0] [kernel-poll]
Eshell V5.4.8  (abort with ^G)
1> error_logger:warning_map().
  warning
2> error_logger:warning_msg("Warnings tagged as: ~p~n", [warning]).

=WARNING REPORT==== 11-Aug-2005::15:31:55 ===
Warnings tagged as: warning
ok
3>
User switch command
   --> q
os$ erl +W e
Erlang (BEAM) emulator version 5.4.8 [hipe] [threads:0] [kernel-poll]
Eshell V5.4.8  (abort with ^G)
1> error_logger:warning_map().
  error
2> error_logger:warning_msg("Warnings tagged as: ~p~n", [error]).

=ERROR REPORT==== 11-Aug-2005::15:31:23 ===
Warnings tagged as: error
ok
```

warning_msg(Format) -> ok
warning_msg(Format, Data) -> ok

Types:

- Format = string()
- Data = list()

Log a standard warning event. The Format and Data arguments are the same as the arguments of io:format/2 in STDLIB.

Error logger forwards the event to Logger, including metadata that allows backwards compatibility with legacy error logger event handlers.

The event is handled by the default Logger handler. The log level can be changed to error or info, see warning_map/0.

These functions are kept for backwards compatibility and must not be used by new code. Use the ?LOG_WARNING macro or logger:warning/1,2,3 instead.

Warning:

If the Unicode translation modifier (t) is used in the format string, all event handlers must ensure that the formatted output is correctly encoded for the I/O device.
warning_report(Report) -> ok
Types:
  Report = report()
Log a standard warning event. Error logger forwards the event to Logger, including metadata that allows backwards compatibility with legacy error logger event handlers.

The event is handled by the default Logger handler. The log level can be changed to error or info, see warning_map/0.

This functions is kept for backwards compatibility and must not be used by new code. Use the ?LOG_WARNING macro or logger:warning/1,2,3 instead.

warning_report(Type, Report) -> ok
Types:
  Type = any()
  Report = report()
Log a user-defined warning event. Error logger forwards the event to Logger, including metadata that allows backwards compatibility with legacy error logger event handlers.

Error logger also adds a domain field with value [Type] to this event's metadata, causing the filters of the default Logger handler to discard the event. A different Logger handler, or an error logger event handler, must be added to handle this event.

The log level can be changed to error or info, see warning_map/0.

It is recommended that Report follows the same structure as for warning_report/1.

This functions is kept for backwards compatibility and must not be used by new code. Use the ?LOG_WARNING macro or logger:warning/1,2,3 instead.

Events
All event handlers added to the error logger must handle the following events. Gleader is the group leader pid of the process that sent the event, and Pid is the process that sent the event.

{error, Gleader, {Pid, Format, Data}}
  Generated when error_msg/1,2 or format is called.
{error_report, Gleader, {Pid, std_error, Report}}
  Generated when error_report/1 is called.
{error_report, Gleader, {Pid, Type, Report}}
  Generated when error_report/2 is called.
{warning_msg, Gleader, {Pid, Format, Data}}
  Generated when warning_msg/1,2 is called if warnings are set to be tagged as warnings.
{warning_report, Gleader, {Pid, std_warning, Report}}
  Generated when warning_report/1 is called if warnings are set to be tagged as warnings.
{warning_report, Gleader, {Pid, Type, Report}}
  Generated when warning_report/2 is called if warnings are set to be tagged as warnings.
{info_msg, Gleader, {Pid, Format, Data}}
  Generated when info_msg/1,2 is called.
error_logger

{info_report, Gleader, {Pid, std_info, Report}}
    Generated when info_report/1 is called.

{info_report, Gleader, {Pid, Type, Report}}
    Generated when info_report/2 is called.

Notice that some system-internal events can also be received. Therefore a catch-all clause last in the definition of the event handler callback function Module:handle_event/2 is necessary. This also applies for Module:handle_info/2, as the event handler must also take care of some system-internal messages.

See Also
gen_event(3), logger(3), log_mf_h(3), kernel(6), sasl(6)
file

Erlang module

This module provides an interface to the file system.

Warning:

File operations are only guaranteed to appear atomic when going through the same file server. A NIF or other OS process may observe intermediate steps on certain operations on some operating systems, eg. renaming an existing file on Windows, or write_file_info/2 on any OS at the time of writing.

Regarding filename encoding, the Erlang VM can operate in two modes. The current mode can be queried using function native_name_encoding/0. It returns latin1 or utf8.

In latin1 mode, the Erlang VM does not change the encoding of filenames. In utf8 mode, filenames can contain Unicode characters greater than 255 and the VM converts filenames back and forth to the native filename encoding (usually UTF-8, but UTF-16 on Windows).

The default mode depends on the operating system. Windows and MacOS X enforce consistent filename encoding and therefore the VM uses utf8 mode.

On operating systems with transparent naming (for example, all Unix systems except MacOS X), default is utf8 if the terminal supports UTF-8, otherwise latin1. The default can be overridden using +fnl (to force latin1 mode) or +fnu (to force utf8 mode) when starting erl.

On operating systems with transparent naming, files can be inconsistently named, for example, some files are encoded in UTF-8 while others are encoded in ISO Latin-1. The concept of raw filenames is introduced to handle file systems with inconsistent naming when running in utf8 mode.

A raw filename is a filename specified as a binary. The Erlang VM does not translate a filename specified as a binary on systems with transparent naming.

When running in utf8 mode, functions list_dir/1 and read_link/1 never return raw filenames. To return all filenames including raw filenames, use functions list_dir_all/1 and read_link_all/1.

See also section Notes About Raw Filenames in the STDLIB User's Guide.

Note:

File operations used to accept filenames containing null characters (integer value zero). This caused the name to be truncated and in some cases arguments to primitive operations to be mixed up. Filenames containing null characters inside the filename are now rejected and will cause primitive file operations fail.

Data Types

deep_list() = [char() | atom() | deep_list()]

fd()

A file descriptor representing a file opened in raw mode.

filename() = string()

See also the documentation of the name_all() type.

filename_all() = string() | binary()

See also the documentation of the name_all() type.
io_device() = pid() | fd()
As returned by open/2; pid() is a process handling I/O-protocols.

name() = string() | atom() | deep_list()

If VM is in Unicode filename mode, string() and char() are allowed to be > 255. See also the documentation of the name_all() type.

name_all() =
  string() | atom() | deep_list() | (RawFilename :: binary())

If VM is in Unicode filename mode, characters are allowed to be > 255. RawFilename is a filename not subject to Unicode translation, meaning that it can contain characters not conforming to the Unicode encoding expected from the file system (that is, non-UTF-8 characters although the VM is started in Unicode filename mode). Null characters (integer value zero) are not allowed in filenames (not even at the end).

posix() =
  eacces | eagain | ebadf | ebadmsg | ebussy | edeadlk |
edeadlock | edquot | eexist | efault | efabs | eftype |
eintr | einval | eio | eisdir | eloop | emfile | emlink |
emultihop | enameerror | enfile | enobufs | enodev |
enolck | enolink | enoent | enomem | enospc | enosr | enostr |
enosys | enotblk | enotdir | enotsup | enxio | epoll | epollwait |
eoverflow | eperm | epipe | erange | erofs | espipe | esrch |
estale | etxtbsy | exdev

An atom that is named from the POSIX error codes used in Unix, and in the runtime libraries of most C compilers.

date_time() = calendar:datetime()
Must denote a valid date and time.

file_info() =
  #file_info{size = integer() >= 0 | undefined,
  type =
    device | directory | other | regular |
    symlink | undefined,
  access =
    read | write | read_write | none | undefined,
  atime =
    file:date_time() |
    integer() >= 0 |
    undefined,
  mtime =
    file:date_time() |
    integer() >= 0 |
    undefined,
  ctime =
    file:date_time() |
    integer() >= 0 |
    undefined,
  mode = integer() >= 0 | undefined,
  links = integer() >= 0 | undefined,
  major_device = integer() >= 0 | undefined,
  minor_device = integer() >= 0 | undefined,
  ino = integer() >= 0 | undefined,
  uid = integer() >= 0 | undefined,
gid = integer() >= 0 | undefined}

location() =
  integer() |
  {bof, Offset :: integer()} |
  {cur, Offset :: integer()} |
  {eof, Offset :: integer()} |
  bof | cur | eof

mode() =
  read | write | append | exclusive | raw | binary |
  {delayed_write,
    Size :: integer() >= 0,
    Delay :: integer() >= 0} |
  delayed_write |
  {read_ahead, Size :: integer() >= 1} |
  read_ahead | compressed |
  {encoding, unicode:encoding()} |
  sync

file_info_option() =
  {time, local} | {time, universal} | {time, posix} | raw

Exports

advise(IoDevice, Offset, Length, Advise) -> ok | {error, Reason}

Types:
  IoDevice = io_device()
  Offset = Length = integer()
  Advise = posix_file_advise()
  Reason = posix() | badarg
  posix_file_advise() =
    normal | sequential | random | no_reuse | will_need | dont_need

advise/4 can be used to announce an intention to access file data in a specific pattern in the future, thus allowing the operating system to perform appropriate optimizations.

On some platforms, this function might have no effect.

allocate(File, Offset, Length) -> ok | {error, posix()}

Types:
  File = io_device()
  Offset = Length = integer() >= 0

allocate/3 can be used to preallocate space for a file.

This function only succeeds in platforms that provide this feature.

change_group(Filename, Gid) -> ok | {error, Reason}

Types:
Filename = name_all()
Gid = integer()
Reason = posix() | badarg
Changes group of a file. See write_file_info/2.

close(IoDevice) -> ok | {error, Reason}
Types:
IoDevice = io_device()
Reason = posix() | badarg | terminated
Closes the file referenced by IoDevice. It mostly returns ok, except for some severe errors such as out of memory. Notice that if option delayed_write was used when opening the file, close/1 can return an old write error and not even try to close the file. See open/2.

consult(Filename) -> {ok,_terms} | {error, Reason}
Types:
Filename = name_all()
Terms = [term()]
Reason = posix() | badarg | terminated | system_limit | {Line :: integer(), Mod :: module(), Term :: term()}
Reads Erlang terms, separated by "," from Filename. Returns one of the following:
{ok, Terms}
The file was successfully read.
{error, atom()}
An error occurred when opening the file or reading it. For a list of typical error codes, see open/2.
{error, {Line, Mod, Term}}
An error occurred when interpreting the Erlang terms in the file. To convert the three-element tuple to an English description of the error, use format_error/1.

Example:

f.txt:  {person, "kalle", 25}.
        {person, "pelle", 30}.

1> file:consult("f.txt").
   {ok, [{person,"kalle",25},{person,"pelle",30}]}  

The encoding of Filename can be set by a comment, as described in epp(3).

copy(Source, Destination) -> {ok, BytesCopied} | {error, Reason}
copy(Source, Destination, ByteCount) ->
   {ok, BytesCopied} | {error, Reason}
Types:
Source = Destination = io_device() | Filename | {Filename, Modes}
Filename = name_all()
Modes = [mode()]
ByteCount = integer() >= 0 | infinity
BytesCopied = integer() >= 0
Reason = posix() | badarg | terminated
Copies ByteCount bytes from Source to Destination. Source and Destination refer to either filenames or IO devices from, for example, open/2. ByteCount defaults to infinity, denoting an infinite number of bytes.
Argument Modes is a list of possible modes, see open/2, and defaults to [].

If both Source and Destination refer to filenames, the files are opened with [read, binary] and [write, binary] prepended to their mode lists, respectively, to optimize the copy.

If Source refers to a filename, it is opened with read mode prepended to the mode list before the copy, and closed when done.

If Destination refers to a filename, it is opened with write mode prepended to the mode list before the copy, and closed when done.

Returns {ok, BytesCopied}, where BytesCopied is the number of bytes that was copied, which can be less than ByteCount if end of file was encountered on the source. If the operation fails, {error, Reason} is returned.

Typical error reasons: as for open/2 if a file had to be opened, and as for read/2 and write/2.

datasync(IoDevice) -> ok | {error, Reason}

Types:

  IoDevice = io_device()
  Reason = posix() | badarg | terminated

Ensures that any buffers kept by the operating system (not by the Erlang runtime system) are written to disk. In many ways it resembles fsync but it does not update some of the metadata of the file, such as the access time. On some platforms this function has no effect.

Applications that access databases or log files often write a tiny data fragment (for example, one line in a log file) and then call fsync() immediately to ensure that the written data is physically stored on the hard disk. Unfortunately, fsync() always initiates two write operations: one for the newly written data and another one to update the modification time stored in the inode. If the modification time is not a part of the transaction concept, fdatasync() can be used to avoid unnecessary inode disk write operations.

Available only in some POSIX systems, this call results in a call to fsync(), or has no effect in systems not providing the fdatasync() syscall.

del_dir(Dir) -> ok | {error, Reason}

Types:

  Dir = name_all()
  Reason = posix() | badarg

Tries to delete directory Dir. The directory must be empty before it can be deleted. Returns ok if successful.

Typical error reasons:

  eacces
       Missing search or write permissions for the parent directories of Dir.
  eexist
       The directory is not empty.
  enoent
       The directory does not exist.
  enotdir
       A component of Dir is not a directory. On some platforms, enoent is returned instead.
  einval
       Attempt to delete the current directory. On some platforms, eacces is returned instead.
del_dir_r(File) -> ok | {error, Reason}
Types:
  File = name_all()
  Reason = posix() | badarg
Deletes file or directory File. If File is a directory, its contents is first recursively deleted. Returns:
ok
  The operation completed without errors.
{error, posix()}
  An error occurred when accessing or deleting File. If some file or directory under File could not be deleted, File cannot be deleted as it is non-empty, and {error, eexist} is returned.

delete(Filename) -> ok | {error, Reason}
Types:
  Filename = name_all()
  Reason = posix() | badarg
Tries to delete file Filename. Returns ok if successful.
Typical error reasons:
enoent
  The file does not exist.
  eaccess
  Missing permission for the file or one of its parents.
eperm
  The file is a directory and the user is not superuser.
enotdir
  A component of the filename is not a directory. On some platforms, enoent is returned instead.
eINVAL
  Filename has an improper type, such as tuple.

Warning:
In a future release, a bad type for argument Filename will probably generate an exception.

eval(Filename) -> ok | {error, Reason}
Types:
  Filename = name_all()
  Reason = posix() | badarg | terminated | system_limit |
{Line :: integer(), Mod :: module(), Term :: term()}

Reads and evaluates Erlang expressions, separated by '.' (or ';', a sequence of expressions is also an expression) from Filename. The result of the evaluation is not returned; any expression sequence in the file must be there for its side effect. Returns one of the following:

ok

The file was read and evaluated.

{error, atom()}

An error occurred when opening the file or reading it. For a list of typical error codes, see open/2.

{error, {Line, Mod, Term}}

An error occurred when interpreting the Erlang expressions in the file. To convert the three-element tuple to an English description of the error, use format_error/1.

The encoding of Filename can be set by a comment, as described in epp(3).

eval(Filename, Bindings) -> ok | {error, Reason}

Types:
Filename = name_all()
Bindings = erl_eval:binding_struct()
Reason =
   posix() |
   badarg | terminated | system_limit |
   {Line :: integer(), Mod :: module(), Term :: term()}

The same as eval/1, but the variable bindings Bindings are used in the evaluation. For information about the variable bindings, see erl_eval(3).

format_error(Reason) -> Chars

Types:
Reason =
   posix() |
   badarg | terminated | system_limit |
   {Line :: integer(), Mod :: module(), Term :: term()}
Chars = string()

Given the error reason returned by any function in this module, returns a descriptive string of the error in English.

get_cwd() -> {ok, Dir} | {error, Reason}

Types:
Dir = filename()
Reason = posix()

Returns {ok, Dir}, where Dir is the current working directory of the file server.

Note:

In rare circumstances, this function can fail on Unix. It can occur if read permission does not exist for the parent directories of the current directory.

A typical error reason:
get_cwd(Drive) -> {ok, Dir} | {error, Reason}

Types:
  Drive = string()
  Dir = filename()
  Reason = posix() | badarg

Returns {ok, Dir} or {error, Reason}, where Dir is the current working directory of the specified drive. Drive is to be of the form "Letter:," for example, "c:.

Returns {error, enotsup} on platforms that have no concept of current drive (Unix, for example).

Typical error reasons:

enotsup
  The operating system has no concept of drives.

eacces
  The drive does not exist.

einval
  The format of Drive is invalid.

list_dir(Dir) -> {ok, Filenames} | {error, Reason}

Types:
  Dir = name_all()
  Filenames = [filename()]
  Reason =
  posix() | badarg |
  {no_translation, Filename :: unicode:latin1_binary()}

Lists all files in a directory, except files with raw filenames. Returns {ok, Filenames} if successful, otherwise {error, Reason}. Filenames is a list of the names of all the files in the directory. The names are not sorted.

Typical error reasons:

eacces
  Missing search or write permissions for Dir or one of its parent directories.

enoent
  The directory does not exist.

{no_translation, Filename}
  Filename is a binary() with characters coded in ISO Latin-1 and the VM was started with parameter +fnue.

list_dir_all(Dir) -> {ok, Filenames} | {error, Reason}

Types:
Dir = name_all()
Filenames = [filename_all()]
Reason = posix() | badarg

Lists all the files in a directory, including files with raw filenames. Returns {ok, Filenames} if successful, otherwise {error, Reason}. Filenames is a list of the names of all the files in the directory. The names are not sorted.

Typical error reasons:

eacces

Missing search or write permissions for Dir or one of its parent directories.

enoent

The directory does not exist.

make_dir(Dir) -> ok | {error, Reason}

Types:

Dir = name_all()
Reason = posix() | badarg

Tries to create directory Dir. Missing parent directories are not created. Returns ok if successful.

Typical error reasons:

eacces

Missing search or write permissions for the parent directories of Dir.

eexist

A file or directory named Dir exists already.

enoent

A component of Dir does not exist.

enospc

No space is left on the device.

enotdir

A component of Dir is not a directory. On some platforms, enoent is returned instead.

make_link(Existing, New) -> ok | {error, Reason}

Types:

Existing = New = name_all()
Reason = posix() | badarg

Makes a hard link from Existing to New on platforms supporting links (Unix and Windows). This function returns ok if the link was successfully created, otherwise {error, Reason}. On platforms not supporting links, {error, enotsup} is returned.

Typical error reasons:

eacces

Missing read or write permissions for the parent directories of Existing or New.
make_symmlink(Existing, New) -> ok | {error, Reason}
Types:
  Existing = New = name_all()
  Reason = posix() | badarg

Creates a symbolic link New to the file or directory Existing on platforms supporting symbolic links (most Unix systems and Windows, beginning with Vista). Existing does not need to exist. Returns ok if the link is successfully created, otherwise {error, Reason}. On platforms not supporting symbolic links, {error, enotsup} is returned.

Typical error reasons:

  eacces
  Missing read or write permissions for the parent directories of Existing or New.

  eexist
  New already exists.

  enotsup
  Symbolic links are not supported on this platform.

  eperm
  User does not have privileges to create symbolic links (SeCreateSymbolicLinkPrivilege on Windows).

native_name_encoding() -> latin1 | utf8

Returns the filename encoding mode. If it is latin1, the system translates no filenames. If it is utf8, filenames are converted back and forth to the native filename encoding (usually UTF-8, but UTF-16 on Windows).

open(File, Modes) -> {ok, IoDevice} | {error, Reason}
Types:
  File = Filename | iodata()
  Filename = name_all()
  Modes = [mode() | ram | directory]
  IoDevice = io_device()
  Reason = posix() | badarg | system_limit

Opens file File in the mode determined by Modes, which can contain one or more of the following options:

  read
  The file, which must exist, is opened for reading.

  write
  The file is opened for writing. It is created if it does not exist. If the file exists and write is not combined with read, the file is truncated.
append

The file is opened for writing. It is created if it does not exist. Every write operation to a file opened with append takes place at the end of the file.

exclusive

The file is opened for writing. It is created if it does not exist. If the file exists, \{error, eexist\} is returned.

**Warning:**

This option does not guarantee exclusiveness on file systems not supporting O_EXCL properly, such as NFS. Do not depend on this option unless you know that the file system supports it (in general, local file systems are safe).

raw

Allows faster access to a file, as no Erlang process is needed to handle the file. However, a file opened in this way has the following limitations:

- The functions in the io module cannot be used, as they can only talk to an Erlang process. Instead, use functions read/2, read_line/1, and write/2.
- Especially if read_line/1 is to be used on a raw file, it is recommended to combine this option with option \{read_ahead, Size\} as line-oriented I/O is inefficient without buffering.
- Only the Erlang process that opened the file can use it.
- A remote Erlang file server cannot be used. The computer on which the Erlang node is running must have access to the file system (directly or through NFS).

binary

Read operations on the file return binaries rather than lists.

\{delayed_write, Size, Delay\}

Data in subsequent write/2 calls is buffered until at least Size bytes are buffered, or until the oldest buffered data is Delay milliseconds old. Then all buffered data is written in one operating system call. The buffered data is also flushed before some other file operation than write/2 is executed.

The purpose of this option is to increase performance by reducing the number of operating system calls. Thus, the write/2 calls must be for sizes significantly less than Size, and not interspersed by too many other file operations.

When this option is used, the result of write/2 calls can prematurely be reported as successful, and if a write error occurs, the error is reported as the result of the next file operation, which is not executed.

For example, when delayed_write is used, after a number of write/2 calls, close/1 can return \{error, enospc\}, as there is not enough space on the disc for previously written data. close/1 must probably be called again, as the file is still open.

delayed_write

The same as \{delayed_write, Size, Delay\} with reasonable default values for Size and Delay (roughly some 64 KB, 2 seconds).

\{read_ahead, Size\}

Activates read data buffering. If read/2 calls are for significantly less than Size bytes, read operations to the operating system are still performed for blocks of Size bytes. The extra data is buffered and returned in subsequent read/2 calls, giving a performance gain as the number of operating system calls is reduced.
The read_ahead buffer is also highly used by function read_line/1 in raw mode, therefore this option is recommended (for performance reasons) when accessing raw files using that function.

If read/2 calls are for sizes not significantly less than, or even greater than Size bytes, no performance gain can be expected.

read_ahead

The same as \{read_ahead, Size\} with a reasonable default value for Size (roughly some 64 KB).

compressed

Makes it possible to read or write gzip compressed files. Option compressed must be combined with read or write, but not both. Notice that the file size obtained with read_file_info/1 does probably not match the number of bytes that can be read from a compressed file.

\{(encoding, Encoding)\}

Makes the file perform automatic translation of characters to and from a specific (Unicode) encoding. Notice that the data supplied to write/2 or returned by read/2 still is byte-oriented; this option denotes only how data is stored in the disk file.

Depending on the encoding, different methods of reading and writing data is preferred. The default encoding of latin1 implies using this module (file) for reading and writing data as the interfaces provided here work with byte-oriented data. Using other (Unicode) encodings makes the io(3) functions get_chars, get_line, and put_chars more suitable, as they can work with the full Unicode range.

If data is sent to an io_device() in a format that cannot be converted to the specified encoding, or if data is read by a function that returns data in a format that cannot cope with the character range of the data, an error occurs and the file is closed.

Allowed values for Encoding:

latin1

The default encoding. Bytes supplied to the file, that is, write/2 are written "as is" on the file. Likewise, bytes read from the file, that is, read/2 are returned "as is". If module io(3) is used for writing, the file can only cope with Unicode characters up to code point 255 (the ISO Latin-1 range).

unicode or utf8

Characters are translated to and from UTF-8 encoding before they are written to or read from the file. A file opened in this way can be readable using function read/2, as long as no data stored on the file lies beyond the ISO Latin-1 range (0..255), but failure occurs if the data contains Unicode code points beyond that range. The file is best read with the functions in the Unicode aware module io(3).

Bytes written to the file by any means are translated to UTF-8 encoding before being stored on the disk file.

utf16 or {utf16,big}

Works like unicode, but translation is done to and from big endian UTF-16 instead of UTF-8.

{utf16,little}

Works like unicode, but translation is done to and from little endian UTF-16 instead of UTF-8.

utf32 or {utf32,big}

Works like unicode, but translation is done to and from big endian UTF-32 instead of UTF-8.

{utf32,little}

Works like unicode, but translation is done to and from little endian UTF-32 instead of UTF-8.
The Encoding can be changed for a file "on the fly" by using function \texttt{io:setopts/2}. So a file can be analyzed in latin1 encoding for, for example, a BOM, positioned beyond the BOM and then be set for the right encoding before further reading. For functions identifying BOMs, see module \texttt{unicode(3)}.

This option is not allowed on raw files.

\texttt{ram}

File must be \texttt{iodata()}. Returns an \texttt{fd()}, which lets module \texttt{file} operate on the data in-memory as if it is a file.

\texttt{sync}

On platforms supporting it, enables the POSIX O\_SYNC synchronous I/O flag or its platform-dependent equivalent (for example, \texttt{FILE\_FLAG\_WRITE\_THROUGH} on Windows) so that writes to the file block until the data is physically written to disk. However, be aware that the exact semantics of this flag differ from platform to platform. For example, none of Linux or Windows guarantees that all file metadata are also written before the call returns. For precise semantics, check the details of your platform documentation. On platforms with no support for POSIX O\_SYNC or equivalent, use of the \texttt{sync} flag causes \texttt{open} to return \{\texttt{error}, enotsup\}.

\texttt{directory}

Allows \texttt{open} to work on directories.

Returns:

\{\texttt{ok, IoDevice}\}

The file is opened in the requested mode. \texttt{IoDevice} is a reference to the file.

\{\texttt{error, Reason}\}

The file cannot be opened.

\texttt{IoDevice} is really the pid of the process that handles the file. This process monitors the process that originally opened the file (the owner process). If the owner process terminates, the file is closed and the process itself terminates too. An \texttt{IoDevice} returned from this call can be used as an argument to the I/O functions (see \texttt{io(3)}).

\begin{tcolorbox}[colback=green!10!white,colframe=green!75!black]
\textbf{Note:}

In previous versions of \texttt{file}, modes were specified as one of the atoms \texttt{read}, \texttt{write}, or \texttt{read\_write} instead of a list. This is still allowed for reasons of backwards compatibility, but is not to be used for new code. Also note that \texttt{read\_write} is not allowed in a mode list.
\end{tcolorbox}

Typical error reasons:

\texttt{enoent}

The file does not exist.

\texttt{eacces}

Missing permission for reading the file or searching one of the parent directories.

\texttt{eisdir}

The named file is a directory.

\texttt{enotdir}

A component of the filename is not a directory, or the filename itself is not a directory if directory mode was specified. On some platforms, \texttt{enoent} is returned instead.
enospc

There is no space left on the device (if write access was specified).

path_consult(Path, Filename) ->
   {ok, Terms, FullName} | {error, Reason}
Types:
   Path = [Dir]
   Dir = Filename = name_all()
   Terms = [term()]
   FullName = filename_all()
   Reason =
      posix() |
      badarg | terminated | system_limit |
      {Line :: integer(), Mod :: module(), Term :: term()}

Searches the path Path (a list of directory names) until the file Filename is found. If Filename is an absolute filename, Path is ignored. Then reads Erlang terms, separated by '.', from the file.

Returns one of the following:

{ok, Terms, FullName}
   The file is successfully read. FullName is the full name of the file.

{error, enoent}
   The file cannot be found in any of the directories in Path.

{error, atom()}
   An error occurred when opening the file or reading it. For a list of typical error codes, see open/2.

{error, {Line, Mod, Term}}
   An error occurred when interpreting the Erlang terms in the file. Use format_error/1 to convert the three-element tuple to an English description of the error.

The encoding of Filename can be set by a comment as described in epp(3).

path_eval(Path, Filename) -> {ok, FullName} | {error, Reason}
Types:
   Path = [Dir :: name_all()]
   Filename = name_all()
   FullName = filename_all()
   Reason =
      posix() |
      badarg | terminated | system_limit |
      {Line :: integer(), Mod :: module(), Term :: term()}

Searches the path Path (a list of directory names) until the file Filename is found. If Filename is an absolute filename, Path is ignored. Then reads and evaluates Erlang expressions, separated by '.' (or ';', a sequence of expressions is also an expression), from the file. The result of evaluation is not returned; any expression sequence in the file must be there for its side effect.

Returns one of the following:
The encoding of `Filename` can be set by a comment as described in `epp(3)`.

```erlang
path_open(Path, Filename, Modes) ->
    {ok, IoDevice, FullName} | {error, Reason}
```

Types:
- `Path = [Dir :: name_all()]`
- `Filename = name_all()`
- `Modes = [mode() | directory]`
- `IoDevice = io_device()`
- `FullName = filename_all()`
- `Reason = posix() | badarg | system_limit`

Searches the path `Path` (a list of directory names) until the file `Filename` is found. If `Filename` is an absolute filename, `Path` is ignored. Then opens the file in the mode determined by `Modes`.

Returns one of the following:
- `{ok, IoDevice, FullName}`
  The file is opened in the requested mode. `IoDevice` is a reference to the file and `FullName` is the full name of the file.
- `{error, enoent}`
  The file cannot be found in any of the directories in `Path`.
- `{error, atom()}`
  The file cannot be opened.

```erlang
path_script(Path, Filename) ->
    {ok, Value, FullName} | {error, Reason}
```

Types:
- `Path = [Dir :: name_all()]`
- `Filename = name_all()`
- `Value = term()`
- `FullName = filename_all()`
- `Reason = posix() | badarg | terminated | system_limit |`
{Line :: integer(), Mod :: module(), Term :: term()}

Searches the path Path (a list of directory names) until the file Filename is found. If Filename is an absolute filename, Path is ignored. Then reads and evaluates Erlang expressions, separated by '.' (or ',', a sequence of expressions is also an expression), from the file.

Returns one of the following:

{ok, Value, FullName}

The file is read and evaluated. FullName is the full name of the file and Value the value of the last expression.

{error, enoent}

The file cannot be found in any of the directories in Path.

{error, atom()}

An error occurred when opening the file or reading it. For a list of typical error codes, see open/2.

{error, {Line, Mod, Term}}

An error occurred when interpreting the Erlang expressions in the file. Use format_error/1 to convert the three-element tuple to an English description of the error.

The encoding of Filename can be set by a comment as described in epp(3).

path_script(Path, Filename, Bindings) ->
             {ok, Value, FullName} | {error, Reason}

Types:
    Path = [Dir :: name_all()]
    Filename = name_all()
    Bindings = erl_eval:binding_struct()
    Value = term()
    FullName = filename_all()
    Reason =
            posix() |
            badarg | terminated | system_limit |
            {Line :: integer(), Mod :: module(), Term :: term()}

The same as path_script/2 but the variable bindings Bindings are used in the evaluation. See erl_eval(3) about variable bindings.

pid2name(Pid) -> {ok, Filename} | undefined

Types:
    Filename = filename_all()
    Pid = pid()

If Pid is an I/O device, that is, a pid returned from open/2, this function returns the filename, or rather:

{ok, Filename}

If the file server of this node is not a slave, the file was opened by the file server of this node (this implies that Pid must be a local pid) and the file is not closed. Filename is the filename in flat string format.

undefined

In all other cases.
position(IoDevice, Location) ->
   {ok, NewPosition} | {error, Reason}

Types:
   IoDevice = io_device()
   Location = location()
   NewPosition = integer()
   Reason = posix() | badarg | terminated

Sets the position of the file referenced by IoDevice to Location. Returns {ok, NewPosition} (as absolute offset) if successful, otherwise {error, Reason}. Location is one of the following:

Offset
   The same as {bof, Offset}.
   (bof, Offset)
      Absolute offset.
   {cur, Offset}
      Offset from the current position.
   {eof, Offset}
      Offset from the end of file.

bof | cur | eof
   The same as above with Offset 0.

Notice that offsets are counted in bytes, not in characters. If the file is opened using some other encoding than latin1, one byte does not correspond to one character. Positioning in such a file can only be done to known character boundaries. That is, to a position earlier retrieved by getting a current position, to the beginning/end of the file or to some other position known to be on a correct character boundary by some other means (typically beyond a byte order mark in the file, which has a known byte-size).

A typical error reason is:

EINVAL

Either Location is illegal, or it is evaluated to a negative offset in the file. Notice that if the resulting position is a negative value, the result is an error, and after the call the file position is undefined.

pread(IoDevice, LocNums) -> {ok, DataL} | eof | {error, Reason}

Types:
   IoDevice = io_device()
   LocNums =
{{Location :: location()}, Number :: integer() >= 0}]
DataL = [Data]
Data = string() | binary() | eof
Reason = posix() | badarg | terminated

Performs a sequence of \texttt{pread/3} in one operation, which is more efficient than calling them one at a time. Returns \{ok, [Data, ...]\} or \{error, Reason\}, where each Data, the result of the corresponding \texttt{pread}, is either a list or a binary depending on the mode of the file, or \texttt{eof} if the requested position is beyond end of file.

As the position is specified as a byte-offset, take special caution when working with files where \texttt{encoding} is set to something else than \texttt{latin1}, as not every byte position is a valid character boundary on such a file.

\texttt{pread(IoDevice, Location, Number) -> ok, Data} | eof | {error, Reason}

Types:
  IoDevice = io\_device()
  Location = location()
  Number = integer() >= 0
  Data = string() | binary()
  Reason = posix() | badarg | terminated

Combines \texttt{position/2} and \texttt{read/2} in one operation, which is more efficient than calling them one at a time. If \texttt{IoDevice} is opened in \texttt{raw} mode, some restrictions apply:
  • Location is only allowed to be an integer.
  • The current position of the file is undefined after the operation.

As the position is specified as a byte-offset, take special caution when working with files where \texttt{encoding} is set to something else than \texttt{latin1}, as not every byte position is a valid character boundary on such a file.

\texttt{pwrite(IoDevice, LocBytes) -> ok | \{error, \{N, Reason\}\}}

Types:
  IoDevice = io\_device()
  LocBytes = [{Location :: location(), Bytes :: iodata()}]
  N = integer() >= 0
  Reason = posix() | badarg | terminated

Performs a sequence of \texttt{pwrite/3} in one operation, which is more efficient than calling them one at a time. Returns \texttt{ok} or \{error, \{N, Reason\}\}, where \texttt{N} is the number of successful writes done before the failure.

When positioning in a file with other \texttt{encoding} than \texttt{latin1}, caution must be taken to set the position on a correct character boundary. For details, see \texttt{position/2}.

\texttt{pwrite(IoDevice, Location, Bytes) -> ok | \{error, Reason\}}

Types:
  IoDevice = io\_device()
  Location = location()
  Bytes = iodata()
  Reason = posix() | badarg | terminated

Combines \texttt{position/2} and \texttt{write/2} in one operation, which is more efficient than calling them one at a time. If \texttt{IoDevice} has been opened in \texttt{raw} mode, some restrictions apply:
• Location is only allowed to be an integer.
• The current position of the file is undefined after the operation.

When positioning in a file with other encoding than latin1, caution must be taken to set the position on a correct character boundary. For details, see position/2.

read(IoDevice, Number) -> {ok, Data} | eof | {error, Reason}
Types:
  IoDevice = io_device() | atom()
  Number = integer() >= 0
  Data = string() | binary()
  Reason =
    posix() |
    badarg | terminated |
    {no_translation, unicode, latin1}

Reads Number bytes/characters from the file referenced by IoDevice. The functions read/2, pread/3, and read_line/1 are the only ways to read from a file opened in raw mode (although they work for normally opened files, too).

For files where encoding is set to something else than latin1, one character can be represented by more than one byte on the file. The parameter Number always denotes the number of characters read from the file, while the position in the file can be moved much more than this number when reading a Unicode file.

Also, if encoding is set to something else than latin1, the read/3 call fails if the data contains characters larger than 255, which is why module io(3) is to be preferred when reading such a file.

The function returns:

{ok, Data}
If the file was opened in binary mode, the read bytes are returned in a binary, otherwise in a list. The list or binary is shorter than the number of bytes requested if end of file was reached.

eof
Returned if Number>0 and end of file was reached before anything at all could be read.

{error, Reason}
An error occurred.

Typical error reasons:

ebadf
The file is not opened for reading.

{no_translation, unicode, latin1}
The file is opened with another encoding than latin1 and the data in the file cannot be translated to the byte-oriented data that this function returns.

read_file(Filename) -> {ok, Binary} | {error, Reason}
Types:
Filename = name_all()
Binary = binary()
Reason = posix() | badarg | terminated | system_limit

Returns {ok, Binary}, where Binary is a binary data object that contains the contents of Filename, or {error, Reason} if an error occurs.

Typical error reasons:

- enoent
  The file does not exist.

- eacces
  Missing permission for reading the file, or for searching one of the parent directories.

- eisdir
  The named file is a directory.

- enotdir
  A component of the filename is not a directory. On some platforms, enoent is returned instead.

- enomem
  There is not enough memory for the contents of the file.


read_file_info(File) -> {ok, FileInfo} | {error, Reason}
read_file_info(File, Opts) -> {ok, FileInfo} | {error, Reason}

Types:
- File = name_all() | io_device()
- Opts = [file_info_option()]
- FileInfo = file_info()
- Reason = posix() | badarg

Retrieves information about a file. Returns {ok, FileInfo} if successful, otherwise {error, Reason}. FileInfo is a record file_info, defined in the Kernel include file file.hrl. Include the following directive in the module from which the function is called:

```erlang
(include_lib("kernel/include/file.hrl").
```

The time type returned in atime, mtime, and ctime is dependent on the time type set in Opts :: {time, Type} as follows:

- local
  Returns local time.

- universal
  Returns universal time.

- posix
  Returns seconds since or before Unix time epoch, which is 1970-01-01 00:00 UTC.

Default is {time, local}.

If the option raw is set, the file server is not called and only information about local files is returned. Note that this will break this module's atomicity guarantees as it can race with a concurrent call to write_file_info/1,2.
This option has no effect when the function is given an I/O device instead of a file name. Use open/2 with the raw mode to obtain a file descriptor first.

Note:

As file times are stored in POSIX time on most OS, it is faster to query file information with option posix.

The record file_info contains the following fields:

- `size = integer() >= 0`
  Size of file in bytes.

- `type = device | directory | other | regular | symlink`
  The type of the file.

- `access = read | write | read_write | none`
  The current system access to the file.

- `atime = date_time() | integer() >= 0`
  The last time the file was read.

- `mtime = date_time() | integer() >= 0`
  The last time the file was written.

- `ctime = date_time() | integer() >=0`
  The interpretation of this time field depends on the operating system. On Unix, it is the last time the file or the inode was changed. In Windows, it is the create time.

- `mode = integer() >= 0`
  The file permissions as the sum of the following bit values:

  - `00400` read permission: owner
  - `00200` write permission: owner
  - `00100` execute permission: owner
  - `00040` read permission: group
  - `00020` write permission: group
  - `00010` execute permission: group
  - `00004` read permission: other
  - `00002` write permission: other
execute permission: other

set user id on execution

set group id on execution

On Unix platforms, other bits than those listed above may be set.

links = integer() >= 0

Number of links to the file (this is always 1 for file systems that have no concept of links).

major_device = integer() >= 0

Identifies the file system where the file is located. In Windows, the number indicates a drive as follows: 0 means A:, 1 means B:, and so on.

minor_device = integer() >= 0

Only valid for character devices on Unix. In all other cases, this field is zero.

inode = integer() >= 0

Gives the inode number. On non-Unix file systems, this field is zero.

uid = integer() >= 0

Indicates the owner of the file. On non-Unix file systems, this field is zero.

gid = integer() >= 0

Gives the group that the owner of the file belongs to. On non-Unix file systems, this field is zero.

Typical error reasons:

eacces

Missing search permission for one of the parent directories of the file.

enoent

The file does not exist.

enotdir

A component of the filename is not a directory. On some platforms, enoent is returned instead.

read_line(IoDevice) -> {ok, Data} | eof | {error, Reason}

Types:

IoDevice = io_device() | atom()
Data = string() | binary()
Reason =
    posix() |
    badarg | terminated |
    {no_translation, unicode, latin1}

Reads a line of bytes/characters from the file referenced by IoDevice. Lines are defined to be delimited by the linefeed (LF, \n) character, but any carriage return (CR, \r) followed by a newline is also treated as a single LF character (the carriage return is silently ignored). The line is returned including the LF, but excluding any CR
immediately followed by an LF. This behaviour is consistent with the behaviour of \texttt{io:get_line/2}. If end of file is reached without any LF ending the last line, a line with no trailing LF is returned.

The function can be used on files opened in \texttt{raw} mode. However, it is inefficient to use it on \texttt{raw} files if the file is not opened with option \texttt{(read_ahead, Size)} specified. Thus, combining \texttt{raw} and \texttt{(read_ahead, Size)} is highly recommended when opening a text file for raw line-oriented reading.

If \texttt{encoding} is set to something else than \texttt{latin1}, the \texttt{read_line/1} call fails if the data contains characters larger than 255, why module \texttt{io(3)} is to be preferred when reading such a file.

The function returns:

\begin{itemize}
  \item \texttt{(ok, Data)}
    \begin{itemize}
      \item One line from the file is returned, including the trailing LF, but with CRLF sequences replaced by a single LF (see above).
      \item If the file is opened in binary mode, the read bytes are returned in a binary, otherwise in a list.
    \end{itemize}
  \item \texttt{eof}
    \begin{itemize}
      \item Returned if end of file was reached before anything at all could be read.
    \end{itemize}
  \item \texttt{(error, Reason)}
    \begin{itemize}
      \item An error occurred.
    \end{itemize}
\end{itemize}

Typical error reasons:

\begin{itemize}
  \item \texttt{ebadf}
    \begin{itemize}
      \item The file is not opened for reading.
    \end{itemize}
  \item \texttt{(no_translation, unicode, latin1)}
    \begin{itemize}
      \item The file is opened with another \texttt{encoding} than \texttt{latin1} and the data on the file cannot be translated to the byte-oriented data that this function returns.
    \end{itemize}
\end{itemize}

\texttt{read_link(Name)} \rightarrow \{\texttt{ok, Filename}\} | \{\texttt{error, Reason}\}

\texttt{read_link_all(Name)} \rightarrow \{\texttt{ok, Filename}\} | \{\texttt{error, Reason}\}
Name = name_all()
Filename = filename_all()
Reason = posix() | badarg

Returns {ok, Filename} if Name refers to a symbolic link or {error, Reason} otherwise. On platforms that do not support symbolic links, the return value is {error, enotsup}.

Notice that Filename can be either a list or a binary.

Typical error reasons:

- einval
  Name does not refer to a symbolic link.

- enoent
  The file does not exist.

- enotsup
  Symbolic links are not supported on this platform.

read_link_info(Name) -> {ok, FileInfo} | {error, Reason}
read_link_info(Name, Opts) -> {ok, FileInfo} | {error, Reason}

Types:
Name = name_all()
Opts = [file_info_option()]
FileInfo = file_info()
Reason = posix() | badarg

Works like read_file_info/1,2 except that if Name is a symbolic link, information about the link is returned in the file_info record and the type field of the record is set to symlink.

If the option raw is set, the file server is not called and only information about local files is returned. Note that this will break this module's atomicity guarantees as it can race with a concurrent call to write_file_info/1,2.

If Name is not a symbolic link, this function returns the same result as read_file_info/1. On platforms that do not support symbolic links, this function is always equivalent to read_file_info/1.

rename(Source, Destination) -> ok | {error, Reason}

Types:
Source = Destination = name_all()
Reason = posix() | badarg

Tries to rename the file Source to Destination. It can be used to move files (and directories) between directories, but it is not sufficient to specify the destination only. The destination filename must also be specified. For example, if bar is a normal file and foo and baz are directories, rename("foo/bar", "baz") returns an error, but rename("foo/bar", "baz/bar") succeeds. Returns ok if it is successful.

Note:

Renaming of open files is not allowed on most platforms (see eacces below).

Typical error reasons:
eacces
   Missing read or write permissions for the parent directories of Source or Destination. On some platforms, this error is given if either Source or Destination is open.

eexist
   Destination is not an empty directory. On some platforms, also given when Source and Destination are not of the same type.

einval
   Source is a root directory, or Destination is a subdirectory of Source.

eisdir
   Destination is a directory, but Source is not.

enoent
   Source does not exist.

enotdir
   Source is a directory, but Destination is not.

exdev
   Source and Destination are on different file systems.

script(Filename) -> {ok, Value} | {error, Reason}
Types:
   Filename = name_all()
   Value = term()
   Reason = posix() | badarg | terminated | system_limit | 
   {Line :: integer(), Mod :: module(), Term :: term()}

Reads and evaluates Erlang expressions, separated by '.' (or ',', a sequence of expressions is also an expression), from the file.

Returns one of the following:

{ok, Value}
   The file is read and evaluated. Value is the value of the last expression.

{error, atom()}
   An error occurred when opening the file or reading it. For a list of typical error codes, see open/2.

{error, {Line, Mod, Term}}
   An error occurred when interpreting the Erlang expressions in the file. Use format_error/1 to convert the three-element tuple to an English description of the error.

The encoding of Filename can be set by a comment as described in epp(3).

script(Filename, Bindings) -> {ok, Value} | {error, Reason}
Types:
Filename = name_all()
Bindings = erl_eval:binding_struct()
Value = term()
Reason =
  posix() |
  badarg | terminated | system_limit |
  {Line :: integer(), Mod :: module(), Term :: term()}

The same as script/1 but the variable bindings Bindings are used in the evaluation. See erl_eval(3) about variable bindings.

sendfile(Filename, Socket) ->
  {ok, integer() >= 0} |
  {error, inet:posix() | closed | badarg | not_owner}

Types:
  Filename = name_all()
  Socket = inet:socket()

Sends the file Filename to Socket. Returns {ok, BytesSent} if successful, otherwise {error, Reason}.

sendfile(RawFile, Socket, Offset, Bytes, Opts) ->
  {ok, integer() >= 0} |
  {error, inet:posix() | closed | badarg | not_owner}

Types:
  RawFile = fd()
  Socket = inet:socket()
  Offset = Bytes = integer() >= 0
  Opts = [sendfile_option()]
  sendfile_option() =
    {chunk_size, integer() >= 0} |
    {use_threads, boolean()}

Sends Bytes from the file referenced by RawFile beginning at Offset to Socket. Returns {ok, BytesSent} if successful, otherwise {error, Reason}. If Bytes is set to 0 all data after the specified Offset is sent.

The file used must be opened using the raw flag, and the process calling sendfile must be the controlling process of the socket. See gen_tcp:controlling_process/2.

If the OS used does not support non-blocking sendfile, an Erlang fallback using read/2 and gen_tcp:send/2 is used.

The option list can contain the following options:

chunk_size
  The chunk size used by the Erlang fallback to send data. If using the fallback, set this to a value that comfortably fits in the systems memory. Default is 20 MB.

set_cwd(Dir) -> ok | {error, Reason}

Types:
Dir = name() | EncodedBinary  
EncodedBinary = binary()  
Reason = posix() | badarg | no_translation

Sets the current working directory of the file server to Dir. Returns ok if successful.

The functions in the module file usually treat binaries as raw filenames, that is, they are passed "as is" even when the encoding of the binary does not agree with native_name_encoding(). However, this function expects binaries to be encoded according to the value returned by native_name_encoding().

Typical error reasons are:

- enoent
  - The directory does not exist.
- enotdir
  - A component of Dir is not a directory. On some platforms, enoent is returned.
- eacces
  - Missing permission for the directory or one of its parents.
- badarg
  - Dir has an improper type, such as tuple.
- no_translation
  - Dir is a binary() with characters coded in ISO-latin-1 and the VM is operating with unicode filename encoding.

**Warning:**

In a future release, a bad type for argument Dir will probably generate an exception.

sync(IoDevice) -> ok | {error, Reason}

Types:

- IoDevice = io_device()
- Reason = posix() | badarg | terminated

Ensures that any buffers kept by the operating system (not by the Erlang runtime system) are written to disk. On some platforms, this function might have no effect.

A typical error reason is:

- enospc
  - Not enough space left to write the file.

truncate(IoDevice) -> ok | {error, Reason}

Types:

- IoDevice = io_device()
- Reason = posix() | badarg | terminated

Truncates the file referenced by IoDevice at the current position. Returns ok if successful, otherwise {error, Reason}. 
write(IoDevice, Bytes) -> ok | {error, Reason}
Types:
  IoDevice = io_device() | atom()
  Bytes = iodata()
  Reason = posix() | badarg | terminated

Writes Bytes to the file referenced by IoDevice. This function is the only way to write to a file opened in raw mode (although it works for normally opened files too). Returns ok if successful, and {error, Reason} otherwise.

If the file is opened with encoding set to something else than latin1, each byte written can result in many bytes being written to the file, as the byte range 0..255 can represent anything between one and four bytes depending on value and UTF encoding type.

Typical error reasons:
ebadf
  The file is not opened for writing.
enospc
  No space is left on the device.

write_file(Filename, Bytes) -> ok | {error, Reason}
Types:
  Filename = name_all()
  Bytes = iodata()
  Reason = posix() | badarg | terminated | system_limit

Writes the contents of the iodata term Bytes to file Filename. The file is created if it does not exist. If it exists, the previous contents are overwritten. Returns ok if successful, otherwise {error, Reason}.

Typical error reasons:
enoeid
  A component of the filename does not exist.
enotdir
  A component of the filename is not a directory. On some platforms, enoent is returned instead.
enospc
  No space is left on the device.
eacces
  Missing permission for writing the file or searching one of the parent directories.
eisdir
  The named file is a directory.
Filename = name_all()
Bytes = iodata()
Modes = [mode()]
Reason = posix() | badarg | terminated | system_limit

Same as write_file/2, but takes a third argument Modes, a list of possible modes, see open/2. The mode flags binary and write are implicit, so they are not to be used.

write_file_info(Filename, FileInfo) -> ok | {error, Reason}
write_file_info(Filename, FileInfo, Opts) -> ok | {error, Reason}

Types:
Filename = name_all()
Opts = [file_info_option()]
FileInfo = file_info()
Reason = posix() | badarg

Changes file information. Returns ok if successful, otherwise {error, Reason}. FileInfo is a record file_info, defined in the Kernel include file file.hrl. Include the following directive in the module from which the function is called:

```
-include_lib("kernel/include/file.hrl").
```

The time type set in atime, mtime, and ctime depends on the time type set in Opts :: {time, Type} as follows:

- local
  Interprets the time set as local.

- universal
  Interprets it as universal time.

- posix
  Must be seconds since or before Unix time epoch, which is 1970-01-01 00:00 UTC.

Default is {time, local}.

If the option raw is set, the file server is not called and only information about local files is returned.

The following fields are used from the record, if they are specified:

- atime = date_time() | integer() >= 0
  The last time the file was read.

- mtime = date_time() | integer() >= 0
  The last time the file was written.

- ctime = date_time() | integer() >= 0
  On Unix, any value specified for this field is ignored (the "ctime" for the file is set to the current time). On Windows, this field is the new creation time to set for the file.

- mode = integer() >= 0
  The file permissions as the sum of the following bit values:

  $00400
  Read permission: owner


8#00200  
Write permission: owner
8#00100  
Execute permission: owner
8#00040  
Read permission: group
8#00020  
Write permission: group
8#00010  
Execute permission: group
8#00004  
Read permission: other
8#00002  
Write permission: other
8#00001  
Execute permission: other
16#800  
Set user id on execution
16#400  
Set group id on execution

On Unix platforms, other bits than those listed above may be set.

uid = integer() >= 0  
Indicates the file owner. Ignored for non-Unix file systems.

gid = integer() >= 0  
Gives the group that the file owner belongs to. Ignored for non-Unix file systems.

Typical error reasons:
eacces  
Missing search permission for one of the parent directories of the file.
enoent  
The file does not exist.
enotdir  
A component of the filename is not a directory. On some platforms, enoent is returned instead.

**POSIX Error Codes**
- eacces - Permission denied
- eagain - Resource temporarily unavailable
- ebadf - Bad file number
- ebusy - File busy
• edquot - Disk quota exceeded
• eexist - File already exists
• efault - Bad address in system call argument
• efbig - File too large
• eintr - Interrupted system call
• einval - Invalid argument
• eio - I/O error
• eisdir - Illegal operation on a directory
• eloop - Too many levels of symbolic links
• emfile - Too many open files
• emlink - Too many links
• enametoolong - Filename too long
• enfile - File table overflow
• enodev - No such device
• enoent - No such file or directory
• enomem - Not enough memory
• enospc - No space left on device
• enotblk - Block device required
• enotdir - Not a directory
• enotsup - Operation not supported
• enxio - No such device or address
• eperm - Not owner
• epipe - Broken pipe
• erofs - Read-only file system
• espipe - Invalid seek
• esrch - No such process
• estate - Stale remote file handle
• exdev - Cross-domain link

Performance
For increased performance, raw files are recommended.

A normal file is really a process so it can be used as an I/O device (see io). Therefore, when data is written to a normal file, the sending of the data to the file process, copies all data that are not binaries. Opening the file in binary mode and writing binaries is therefore recommended. If the file is opened on another node, or if the file server runs as slave to the file server of another node, also binaries are copied.

Note:
Raw files use the file system of the host machine of the node. For normal files (non-raw), the file server is used to find the files, and if the node is running its file server as slave to the file server of another node, and the other node runs on some other host machine, they can have different file systems. However, this is seldom a problem.

open/2 can be given the options delayed_write and read_ahead to turn on caching, which will reduce the number of operating system calls and greatly improve performance for small reads and writes. However, the overhead
won’t disappear completely and it’s best to keep the number of file operations to a minimum. As a contrived example, the following function writes 4MB in 2.5 seconds when tested:

```erlang
create_file_slow(Name) ->
    {ok, Fd} = file:open(Name, [raw, write, delayed_write, binary]),
    create_file_slow_1(Fd, 4 bsl 20),
    file:close(Fd).
create_file_slow_1(_Fd, 0) -> ok;
create_file_slow_1(Fd, M) ->
    ok = file:write(Fd, <<0>>),
    create_file_slow_1(Fd, M - 1).
```

The following functionally equivalent code writes 128 bytes per call to write/2 and so does the same work in 0.08 seconds, which is roughly 30 times faster:

```erlang
create_file(Name) ->
    {ok, Fd} = file:open(Name, [raw, write, delayed_write, binary]),
    create_file_1(Fd, 4 bsl 20),
    file:close(Fd),
    ok.
create_file_1(_Fd, 0) -> ok;
create_file_1(Fd, M) when M >= 128 ->
    ok = file:write(Fd, <<0:(128)/unit:8>>),
    create_file_1(Fd, M - 128);
create_file_1(Fd, M) ->
    ok = file:write(Fd, <<0:(M)/unit:8>>),
    create_file_1(Fd, M - 1).
```

When writing data it’s generally more efficient to write a list of binaries rather than a list of integers. It is not needed to flatten a deep list before writing. On Unix hosts, scatter output, which writes a set of buffers in one operation, is used when possible. In this way `write(FD, [Bin1, Bin2 | Bin3])` writes the contents of the binaries without copying the data at all, except for perhaps deep down in the operating system kernel.

**Warning:**

If an error occurs when accessing an open file with module io, the process handling the file exits. The dead file process can hang if a process tries to access it later. This will be fixed in a future release.

**See Also**

filename(3)
This module provides functions for communicating with sockets using the SCTP protocol. The implementation assumes that the OS kernel supports SCTP (RFC 2960) through the user-level Sockets API Extensions.

During development, this implementation was tested on:

- Linux Fedora Core 5.0 (kernel 2.6.15-2054 or later is needed)
- Solaris 10, 11

During OTP adaptation it was tested on:

- SUSE Linux Enterprise Server 10 (x86_64) kernel 2.6.16.27-0.6-smp, with lksctp-tools-1.0.6
- Briefly on Solaris 10
- SUSE Linux Enterprise Server 10 Service Pack 1 (x86_64) kernel 2.6.16.54-0.2.3-smp with lksctp-tools-1.0.7
- FreeBSD 8.2

This module was written for one-to-many style sockets (type seqpacket). With the addition of peeloff/2, one-to-one style sockets (type stream) were introduced.

Record definitions for this module can be found using:

```erl
-include_lib("kernel/include/inet_sctp.hrl").
```

These record definitions use the "new" spelling 'adaptation', not the deprecated 'adaption', regardless of which spelling the underlying C API uses.

**Data Types**

**assoc_id()**

An opaque term returned in, for example, #sctp_paddr_change{}, which identifies an association for an SCTP socket. The term is opaque except for the special value 0, which has a meaning such as "the whole endpoint" or "all future associations".

```erl
option() =
  [{active, true | false | once | -32768..32767}] |
  [{buffer, integer()} => 0] |
  [{doroute, boolean()}] |
  [{high_msgq_watermark, integer()} => 1] |
  [{linger, {boolean(), integer()} => 0}] |
  [{low_msgq_watermark, integer()} => 1] |
  [{mode, list | binary}] |
  [list | binary] |
  [{priority, integer()} => 0] |
  [{recbuf, integer()} => 0] |
  [{reuseaddr, boolean()}] |
  [{ipv6_v6only, boolean()}] |
  [{sctp_adaptation_layer, #sctp_setadaptation{}}] |
  [{sctp_associnfo, #sctp_assocparams{}}] |
  [{sctp_autoclose, integer()} => 0] |
  [{sctp_default_send_param, #sctp_sndrcvinfo{}}] |
  [{sctp_delayed_ack_time, #sctp_assoc_value{}}]
```

...
One of the SCTP Socket Options.

```erlang
option_name() =
  active | buffer | dontroute | high_msgq_watermark | linger |
low_msgq_watermark | mode | priority | recbuf | reuseaddr |
ipv6_v6only | sctp_adaptation_layer | sctp_associnfo |
sctp_autoclose | sctp_default_send_param |
sctp_delayed_ack_time | sctp_disable_fragments | sctp_events |
sctp_get_peer_addr_info | sctp_i_want_mapped_v4_addr |
sctp_initmsg | sctp_maxseg | sctp_nodelay |
sctp_peer_addr_params | sctp_primary_addr | sctp_rtoinfo |
sctp_set_peer_primary_addr | sctp_setpeerprim |
sctp_status | sndbuf | tclass | ttl |
recvtos, boolean() |
recvtclass, boolean() |
recvttl, boolean()
```

`sctp_socket()`
Socket identifier returned from `open/*`.

**Exports**

```erlang
abort(Socket, Assoc) -> ok | {error, inet:posix()}
```

Types:
- `Socket = sctp_socket()`
- `Assoc = #sctp_assoc_change{}`

Abnormally terminates the association specified by `Assoc`, without flushing of unsent data. The socket itself remains open. Other associations opened on this socket are still valid, and the socket can be used in new associations.

```erlang
close(Socket) -> ok | {error, inet:posix()}
```

Types:
Socket = sctp_socket()

Closes the socket and all associations on it. The unsent data is flushed as in `eof/2`. The `close/1` call is blocking or otherwise depending of the value of the `linger` socket option. If `close` does not linger or linger time-out expires, the call returns and the data is flushed in the background.

```erlang
connect(Socket, Addr, Port, Opts) ->
    {ok, #sctp_assoc_change{state = comm_up}} |
    {error, #sctp_assoc_change{state = cant_assoc}} |
    {error, inet:posix()}
```

Types:
- `Socket` = `sctp_socket()`
- `Addr` = `inet:ip_address()` | `inet:hostname()`
- `Port` = `inet:port_number()`
- `Opts` = `[Opt :: option()]`

Same as `connect(Socket, Addr, Port, Opts, infinity)`.

```erlang
connect(Socket, Addr, Port, Opts, Timeout) ->
    {ok, #sctp_assoc_change{state = comm_up}} |
    {error, #sctp_assoc_change{state = cant_assoc}} |
    {error, inet:posix()}
```

Types:
- `Socket` = `sctp_socket()`
- `Addr` = `inet:ip_address()` | `inet:hostname()`
- `Port` = `inet:port_number()`
- `Opts` = `[Opt :: option()]`
- `Timeout` = `timeout()`

Establishes a new association for socket `Socket`, with the peer (SCTP server socket) specified by `Addr` and `Port`. `Timeout`, is expressed in milliseconds. A socket can be associated with multiple peers.

**Warning:**

Using a value of `Timeout` less than the maximum time taken by the OS to establish an association (around 4.5 minutes if the default values from RFC 4960 are used), can result in inconsistent or incorrect return values. This is especially relevant for associations sharing the same `Socket` (that is, source address and port), as the controlling process blocks until `connect/*` returns. `connect_init/*` provides an alternative without this limitation.

The result of `connect/*` is an `#sctp_assoc_change()` event that contains, in particular, the new Association ID:

```erlang
#sctp_assoc_change{
    state = atom(),
    error = integer(),
    outbound_streams = integer(),
    inbound_streams = integer(),
    assoc_id = assoc_id()
}
```

The number of outbound and inbound streams can be set by giving an `sctp_initmsg` option to `connect` as in:
All options Opt are set on the socket before the association is attempted. If an option record has undefined field values, the options record is first read from the socket for those values. In effect, Opt option records only define field values to change before connecting.

The returned outbound Streams and inbound Streams are the stream numbers on the socket. These can be different from the requested values (OutStreams and MaxInStreams, respectively) if the peer requires lower values.

state can have the following values:

comm_up
Association is successfully established. This indicates a successful completion of connect.

cant_assoc
The association cannot be established (connect/* failure).

Other states do not normally occur in the output from connect/*. Rather, they can occur in #sctp_assoc_change() events received instead of data in recv/* calls. All of them indicate losing the association because of various error conditions, and are listed here for the sake of completeness:

comm_lost
restart
shutdown_comp

Field error can provide more detailed diagnostics. The error field value can be converted into a string using error_string/1.

connect_init(Socket, Addr, Port, Opts) ->
ok | {error, inet:posix()}

Types:
Socket = sctp_socket()
Addr = inet:ip_address() | inet:hostname()
Port = inet:port_number()
Opts = [option()]

Same as connect_init(Socket, Addr, Port, Opts, infinity).

connect_init(Socket, Addr, Port, Opts, Timeout) ->
ok | {error, inet:posix()}

Types:
Socket = sctp_socket()
Addr = inet:ip_address() | inet:hostname()
Port = inet:port_number()
Opts = [option()]
Timeout = timeout()

Initiates a new association for socket Socket, with the peer (SCTP server socket) specified by Addr and Port.

The fundamental difference between this API and connect/* is that the return value is that of the underlying OS connect(2) system call. If ok is returned, the result of the association establishment is received by the calling
process as an \#sctp_assoc_change{} event. The calling process must be prepared to receive this, or poll for it using recv/*, depending on the value of the active option.

The parameters are as described in connect/*, except the Timeout value.

The timer associated with Timeout only supervises IP resolution of Addr.

controlling_process(Socket, Pid) -> ok | {error, Reason}

Types:
  Socket = sctp_socket()
  Pid = pid()
  Reason = closed | not_owner | badarg | inet:posix()

Assigns a new controlling process Pid to Socket. Same implementation as gen_udp:controlling_process/2.

eof(Socket, Assoc) -> ok | {error, Reason}

Types:
  Socket = sctp_socket()
  Assoc = \#sctp_assoc_change{}
  Reason = term()

Gracefully terminates the association specified by Assoc, with flushing of all unsent data. The socket itself remains open. Other associations opened on this socket are still valid. The socket can be used in new associations.

error_string(ErrorNumber) -> ok | string() | unknown_error

Types:
  ErrorNumber = integer()

Translates an SCTP error number from, for example, \#sctp_remote_error{} or \#sctp_send_failed{} into an explanatory string, or one of the atoms ok for no error or undefined for an unrecognized error.

listen(Socket, IsServer) -> ok | {error, Reason}

listen(Socket, Backlog) -> ok | {error, Reason}

Types:
  Socket = sctp_socket()
  Backlog = integer()
  Reason = term()

Sets up a socket to listen on the IP address and port number it is bound to.

For type seqpacket, sockets (the default) IsServer must be true or false. In contrast to TCP, there is no listening queue length in SCTP. If IsServer is true, the socket accepts new associations, that is, it becomes an SCTP server socket.

For type stream, sockets Backlog define the backlog queue length just like in TCP.
open() -> {ok, Socket} | {error, inet:posix()}
open(Port) -> {ok, Socket} | {error, inet:posix()}
open(Opts) -> {ok, Socket} | {error, inet:posix()}
open(Port, Opts) -> {ok, Socket} | {error, inet:posix()}

Types:

    Opts = [Opt]
    Opt =
        {ip, IP} |
        {ifaddr, IP} |
        inet:address_family() |
        {port, Port} |
        {type, SockType} |
        option()
    IP = inet:ip_address() | any | loopback
    Port = inet:port_number()
    SockType = seqpacket | stream
    Socket = sctp_socket()

Creates an SCTP socket and binds it to the local addresses specified by all \(\{\text{ip}, \text{IP}\}\) (or synonymously \(\{\text{ifaddr}, \text{IP}\}\)) options (this feature is called SCTP multi-homing). The default \(\text{IP}\) and \(\text{Port}\) are any and 0, meaning bind to all local addresses on any free port.

Other options:

inet6

    Sets up the socket for IPv6.

inet

    Sets up the socket for IPv4. This is the default.

A default set of socket options is used. In particular, the socket is opened in binary and passive mode, with SockType \text{seqpacket}, and with reasonably large kernel and driver buffers.

If the socket is in passive mode data can be received through the \text{recv/1,2} calls.

If the socket is in active mode data received data is delivered to the controlling process as messages:

\[
\text{\{sctp, Socket, FromIP, FromPort, \{AncData, Data\}\}}
\]

See \text{recv/1,2} for a description of the message fields.

Note:

This message format unfortunately differs slightly from the \text{gen_udp} message format with ancillary data, and from the \text{recv/1,2} return tuple format.

peeloff(Socket, Assoc) -> {ok, NewSocket} | {error, Reason}

Types:
gen_sctp

Socket = sctp_socket()
Assoc = #sctp_assoc_change{} | assoc_id()
NewSocket = sctp_socket()
Reason = term()

Branches off an existing association Assoc in a socket Socket of type seqpacket (one-to-many style) into a new socket NewSocket of type stream (one-to-one style).

The existing association argument Assoc can be either a #sctp_assoc_change{} record as returned from, for example, recv/*, connect/*, or from a listening socket in active mode. It can also be just the field assoc_id integer from such a record.

recv(Socket) ->
    {ok, {FromIP, FromPort, AncData, Data}} | {error, Reason}
recv(Socket, Timeout) ->
    {ok, {FromIP, FromPort, AncData, Data}} | {error, Reason}

Types:
Socket = sctp_socket()
Timeout = timeout()
FromIP = inet:ip_address()
FromPort = inet:port_number()
AncData = [#sctp_sndrcvinfo{} | inet:ancillary_data()]
Data =
    binary() | string() |
    #sctp_sndrcvinfo{} |
    #sctp_assoc_change{} |
    #sctp_paddr_change{} |
    #sctp_adaptation_event{}
Reason =
    inet:posix() |
    #sctp_send_failed{} |
    #sctp_paddr_change{} |
    #sctp_pdapi_event{} |
    #sctp_remote_error{} |
    #sctp_shutdown_event{}

Receives the Data message from any association of the socket. If the receive times out, {error, timeout} is returned. The default time-out is infinity. FromIP and FromPort indicate the address of the sender.

AncData is a list of ancillary data items that can be received along with the main Data. This list can be empty, or contain a single #sctp_sndrcvinfo{} record if receiving of such ancillary data is enabled (see option sctp_events). It is enabled by default, as such ancillary data provides an easy way of determining the association and stream over which the message is received. (An alternative way is to get the association ID from FromIP and FromPort using socket option sctp_get_peer_addr_info, but this does still not produce the stream number).

AncData may also contain ancillary data from the socket options recvtos, recv_class or recv_ttl, if that is supported by the platform for the socket.

The Data received can be a binary() or a list() of bytes (integers in the range 0 through 255) depending on the socket mode, or an SCTP event.

Possible SCTP events:
- `#sctp_sndrcvinfo()`
- `#sctp_assoc_change()`

```erlang
define(#sctp_paddr_change{}
  addr = {ip_address(), port()},
  state = atom(),
  error = integer(),
  assoc_id = assoc_id()
)
```

Indicates change of the status of the IP address of the peer specified by `addr` within association `assoc_id`. Possible values of `state` (mostly self-explanatory) include:

- `addr_unreachable`
- `addr_available`
- `addr_removed`
- `addr_added`
- `addr_made_prim`
- `addr_confirmed`

In case of an error (for example, `addr_unreachable`), field `error` provides more diagnostics. In such cases, event `#sctp_paddr_change()` is automatically converted into an error term returned by `recv/1`. The error field value can be converted into a string using `error_string/1`.

- `#sctp_send_failed{`
  flags = true | false,
  error = integer(),
  info = #sctp_sndrcvinfo{},
  assoc_id = assoc_id(),
  data = binary()
}
```

The sender can receive this event if a send operation fails.

**flags**

A Boolean specifying if the data has been transmitted over the wire.

**error**

Provides extended diagnostics, use `error_string/1`.

**info**

The original `#sctp_sndrcvinfo()` record used in the failed send/*.

**data**

The whole original data chunk attempted to be sent.

In the current implementation of the Erlang/SCTP binding, this event is internally converted into an error term returned by `recv/*`.

- `#sctp_adaptation_event{`
  adaptation_ind = integer(),
  assoc_id = assoc_id()
}
```

Delivered when a peer sends an adaptation layer indication parameter (configured through option `sctp_adaptation_layer`). Notice that with the current implementation of the Erlang/SCTP binding, this event is disabled by default.
A partial delivery failure. In the current implementation of the Erlang/SCTP binding, this event is internally converted into an error term returned by recv/*.

send(Socket, SndRcvInfo, Data) -> ok | {error, Reason}

Types:
  Socket = sctp_socket()
  SndRcvInfo = #sctp_sndrcvinfo{}
  Data = binary() | iolist()
  Reason = term()

Sends the Data message with all sending parameters from a #sctp_sndrcvinfo{} record. This way, the user can specify the PPID (passed to the remote end) and context (passed to the local SCTP layer), which can be used, for example, for error identification. However, such a fine level of user control is rarely required. The function send/4 is sufficient for most applications.

send(Socket, Assoc, Stream, Data) -> ok | {error, Reason}

Types:
  Socket = sctp_socket()
  Assoc = #sctp_assoc_change{} | assoc_id()
  Stream = integer()
  Data = binary() | iolist()
  Reason = term()

Sends a Data message over an existing association and specified stream.

SCTP Socket Options

The set of admissible SCTP socket options is by construction orthogonal to the sets of TCP, UDP, and generic inet options. Only options listed here are allowed for SCTP sockets. Options can be set on the socket using open/1,2 or inet:setopts/2, retrieved using inet:getopts/2. Options can be changed when calling connect/4,5.

(mode, list|binary) or just list or binary

  Determines the type of data returned from recv/1,2.

  {active, true|false|once|N}

    • If false (passive mode, the default), the caller must do an explicit recv call to retrieve the available data from the socket.
    • If true|once|N (active modes) received data or events are sent to the owning process. See open/0..2 for the message format.
    • If true (full active mode) there is no flow control.

Note:

Note that this can cause the message queue to overflow causing for example the virtual machine to run out of memory and crash.
• **If once**, only one message is automatically placed in the message queue, and after that the mode is automatically reset to passive. This provides flow control and the possibility for the receiver to listen for its incoming SCTP data interleaved with other inter-process messages.

• **If active** is specified as an integer \( N \) in the range -32768 to 32767 (inclusive), that number is added to the socket's counting of data messages to be delivered to the controlling process. If the result of the addition is negative, the count is set to 0. Once the count reaches 0, either through the delivery of messages or by being explicitly set with `inet:setopts/2`, the socket mode is automatically reset to passive({active, false}). When a socket in this active mode transitions to passive mode, the message `{sctp_passive, Socket}` is sent to the controlling process to notify it that if it wants to receive more data messages from the socket, it must call `inet:setopts/2` to set the socket back into an active mode.

\[
\{\text{tos, integer()}
\}
\]

Sets the Type-Of-Service field on the IP datagrams that are sent, to the specified value. This effectively determines a prioritization policy for the outbound packets. The acceptable values are system-dependent.

\[
\{\text{priority, integer()}
\}
\]

A protocol-independent equivalent of `tos` above. Setting priority implies setting `tos` as well.

\[
\{\text{dontroute, true|false}
\}
\]

Defaults to false. If `true`, the kernel does not send packets through any gateway, only sends them to directly connected hosts.

\[
\{\text{reuseaddr, true|false}
\}
\]

Defaults to false. If true, the local binding address \( \{\text{IP,Port}\} \) of the socket can be reused immediately. No waiting in state \text{CLOSE\_WAIT} is performed (can be required for high-throughput servers).

\[
\{\text{sndbuf, integer()}
\}
\]

The size, in bytes, of the OS kernel send buffer for this socket. Sending errors would occur for datagrams larger than `val(sndbuf)`. Setting this option also adjusts the size of the driver buffer (see `buffer` above).

\[
\{\text{recbuf, integer()}
\}
\]

The size, in bytes, of the OS kernel receive buffer for this socket. Sending errors would occur for datagrams larger than `val(recbuf)`. Setting this option also adjusts the size of the driver buffer (see `buffer` above).

\[
\{\text{sctp_module, module()}
\}
\]

Overrides which callback module is used. Defaults to `inet_sctp` for IPv4 and `inet6_sctp` for IPv6.

\[
\{\text{sctp_rtoinfo, #sctp_rtoinfo{}{}
\}
\]

Determines retransmission time-out parameters, in milliseconds, for the association(s) specified by `assoc_id`. \( \text{assoc_id} = 0 \) (default) indicates the whole endpoint. See [RFC 2960](https://tools.ietf.org/rfc/rfc2960.txt) and [Sockets API Extensions for SCTP](http://datatracker.ietf.org/doc/rfc2960) for the exact semantics of the field values.
{sctp_associnfo, #sctp_assocparams{}}

{sictp_assocparams,
assoc_id = assoc_id(),
asocmaxrxt = integer(),
number_peer_destinations = integer(),
peer_rwnd = integer(),
local_rwnd = integer(),
cookie_life = integer()}

Determines association parameters for the association(s) specified by assoc_id.

assoc_id = 0 (default) indicates the whole endpoint. See Sockets API Extensions for SCTP for the discussion of their semantics. Rarely used.

{sctp_initmsg, #sctp_initmsg{}}

{sictp_initmsg,
num_ostreams = integer(),
max_instreams = integer(),
max_attempts = integer(),
max_init_timeo = integer()}

Determines the default parameters that this socket tries to negotiate with its peer while establishing an association with it. It is to be set after open/* but before the first connect/*. #sctp_initmsg{} can also be used as ancillary data with the first call of send/* to a new peer (when a new association is created).

num_ostreams
Number of outbound streams

max_instreams
Maximum number of inbound streams

max_attempts
Maximum retransmissions while establishing an association

max_init_timeo
Time-out, in milliseconds, for establishing an association

{sctp_autoclose, integer() >= 0}

Determines the time, in seconds, after which an idle association is automatically closed. 0 means that the association is never automatically closed.

{sctp_nodelay, true|false}

Turns on/off the Nagle algorithm for merging small packets into larger ones. This improves throughput at the expense of latency.

{sctp_disable_fragments, true|false}

If true, induces an error on an attempt to send a message larger than the current PMTU size (which would require fragmentation/reassembling). Notice that message fragmentation does not affect the logical atomicity of its delivery; this option is provided for performance reasons only.

{sctp_i_want_mapped_v4_addr, true|false}

Turns on/off automatic mapping of IPv4 addresses into IPv6 ones (if the socket address family is AF_INET6).

{sctp_maxseg, integer()}

Determines the maximum chunk size if message fragmentation is used. If 0, the chunk size is limited by the Path MTU only.
{sctp_primary_addr, #sctp_prim{}}

```erlang
#sctp_prim(
    assoc_id = assoc_id(),
    addr = {IP, Port}
)
IP = ip_address()
Port = port_number()
```

For the association specified by `assoc_id`, `{IP, Port}` must be one of the peer addresses. This option determines that the specified address is treated by the local SCTP stack as the primary address of the peer.

{sctp_set_peer_primary_addr, #sctp_setpeerprim{}}

```erlang
#sctp_setpeerprim(
    assoc_id = assoc_id(),
    addr = {IP, Port}
)
IP = ip_address()
Port = port_number()
```

When set, informs the peer to use `{IP, Port}` as the primary address of the local endpoint for the association specified by `assoc_id`.

{sctp_adaptation_layer, #sctp_setadaptation{}}

```erlang
#sctp_setadaptation(
    adaptation_ind = integer()
)
```

When set, requests that the local endpoint uses the value specified by `adaptation_ind` as the Adaptation Indication parameter for establishing new associations. For details, see RFC 2960 and Sockets API Extensions for SCTP.

{sctp_peer_addr_params, #sctp_paddrparams{}}

```erlang
#sctp_paddrparams(
    assoc_id = assoc_id(),
    address = {IP, Port},
    hbinterval = integer(),
    pathmaxrxt = integer(),
    pathmtu = integer(),
    sackdelay = integer(),
    flags = list()
)
IP = ip_address()
Port = port_number()
```

Determines various per-address parameters for the association specified by `assoc_id` and the peer address `address` (the SCTP protocol supports multi-homing, so more than one address can correspond to a specified association).

- `hbinterval`  
  Heartbeat interval, in milliseconds

- `pathmaxrxt`  
  Maximum number of retransmissions before this address is considered unreachable (and an alternative address is selected)
pathmtu
    Fixed Path MTU, if automatic discovery is disabled (see flags below)
sackdelay
    Delay, in milliseconds, for SAC messages (if the delay is enabled, see flags below)
flags
    The following flags are available:
    hb_enable
        Enables heartbeat
    hb_disable
        Disables heartbeat
    hb_demand
        Initiates heartbeat immediately
    pmtud_enable
        Enables automatic Path MTU discovery
    pmtud_disable
        Disables automatic Path MTU discovery
    sackdelay_enable
        Enables SAC delay
    sackdelay_disable
        Disables SAC delay
{sctp_default_send_param, #sctp_sndrcvinfo{}}

#sctp_sndrcvinfo(
    stream = integer(),
    ssn    = integer(),
    flags  = list(),
    ppid   = integer(),
    context = integer(),
    timetolive = integer(),
    tsn    = integer(),
    cumtsn = integer(),
    assoc_id = assoc_id()
)
**abort**
Aborts the current association without flushing any unsent data

**eof**
Gracefully shuts down the current association, with flushing of unsent data

Other fields are rarely used. For complete information, see RFC 2960 and Sockets API Extensions for SCTP.

```
{sctp_events, #sctp_event_subscribe{}}
```

```
#sctp_event_subscribe{
  data_io_event = true | false,
  association_event = true | false,
  address_event = true | false,
  send_failure_event = true | false,
  peer_error_event = true | false,
  shutdown_event = true | false,
  partial_delivery_event = true | false,
  adaptation_layer_event = true | false
}
```

This option determines which SCTP Events are to be received (through `recv/*`) along with the data. The only exception is `data_io_event`, which enables or disables receiving of `#sctp_sndrcvinfo()` ancillary data, not events. By default, all flags except `adaptation_layer_event` are enabled, although `sctp_data_io_event` and `association_event` are used by the driver itself and not exported to the user level.

```
{sctp_delayed_ack_time, #sctp_assoc_value{}}
```

```
#sctp_assoc_value{
  assoc_id = assoc_id(),
  assoc_value = integer()
}
```

Rarely used. Determines the ACK time (specified by `assoc_value`, in milliseconds) for the specified association or the whole endpoint if `assoc_value = 0` (default).

```
{sctp_status, #sctp_status{}}
```

```
#sctp_status{
  assoc_id = assoc_id(),
  state = atom(),
  rwnd = integer(),
  unackdata = integer(),
  penddata = integer(),
  instrms = integer(),
  outstrms = integer(),
  fragmentation_point = integer(),
  primary = #sctp_paddrinfo{}
}
```

This option is read-only. It determines the status of the SCTP association specified by `assoc_id`. The following are the possible values of `state` (the state designations are mostly self-explanatory):

```
sctp_state_empty
  Default. Means that no other state is active.
sctp_state_closed
sctp_state_cookie_wait
sctp_state_cookie_echoed
```
sctp_state_established
sctp_state_shutdown_pending
sctp_state_shutdown_sent
sctp_state_shutdown_received
sctp_state_shutdown_ack_sent

Semantics of the other fields:
sstat_rwnd
   Current receiver window size of the association
sstat_unackdata
   Number of unacked data chunks
sstat_penddata
   Number of data chunks pending receipt
sstat_instrms
   Number of inbound streams
sstat_outstrms
   Number of outbound streams
sstat_fragmentation_point
   Message size at which SCTP fragmentation occurs
sstat_primary
   Information on the current primary peer address (see below for the format of #sctp_paddrinfo{})

{sctp_get_peer_addr_info, #sctp_paddrinfo{}}

#sctp_paddrinfo{
   assoc_id  = assoc_id(),
   address   = {IP, Port},
   state     = inactive | active | unconfirmed,
   cwnd      = integer(),
   srtt      = integer(),
   rto       = integer(),
   mtu       = integer()
}

IP = ip_address()
Port = port_number()

This option is read-only. It determines the parameters specific to the peer address specified by address within the association specified by assoc_id. Field address must be set by the caller; all other fields are filled in on return. If assoc_id = 0 (default), the address is automatically translated into the corresponding association ID. This option is rarely used. For the semantics of all fields, see RFC 2960 and Sockets API Extensions for SCTP.

SCTP Examples

Example of an Erlang SCTP server that receives SCTP messages and prints them on the standard output:
Example of an Erlang SCTP client interacting with the above server. Notice that in this example the client creates an association with the server with 5 outbound streams. Therefore, sending of "Test 0" over stream 0 succeeds, but sending of "Test 5" over stream 5 fails. The client then aborts the association, which results in that the corresponding event is received on the server side.
A simple Erlang SCTP client that uses the connect_init API:
-module(ex3).
-export([client/4]).
-include_lib("kernel/include/inet.hrl").
-include_lib("kernel/include/inet_sctp.hrl").

client(Peer1, Port1, Peer2, Port2)
  when is_tuple(Peer1), is_integer(Port1), is_tuple(Peer2), is_integer(Port2) ->
  {ok, S} = gen_sctp:open(),
  SctpInitMsgOpt = {sctp_initmsg,#sctp_initmsg{num_ostreams=5}},
  ActiveOpt = {active, true},
  Opts = [SctpInitMsgOpt, ActiveOpt],
  ok = gen_sctp:connect(S, Peer1, Port1, Opts),
  ok = gen_sctp:connect(S, Peer2, Port2, Opts),
  io:format("Connections initiated~n", []),
  client_loop(S, Peer1, Port1, undefined, Peer2, Port2, undefined).

client_loop(S, Peer1, Port1, AssocId1, Peer2, Port2, AssocId2) ->
  receive
    {sctp, S, Peer1, Port1, {_Anc, SAC}}
    when is_record(SAC, sctp_assoc_change), AssocId1 == undefined ->
      io:format("Association 1 connect result: ~p. AssocId: ~p~n",
                 [SAC#sctp_assoc_change.state,
                  SAC#sctp_assoc_change.assoc_id]),
      client_loop(S, Peer1, Port1, SAC#sctp_assoc_change.assoc_id, Peer2, Port2, AssocId2);
    {sctp, S, Peer2, Port2, {_Anc, SAC}}
    when is_record(SAC, sctp_assoc_change), AssocId2 == undefined ->
      io:format("Association 2 connect result: ~p. AssocId: ~p~n",
                 [SAC#sctp_assoc_change.state, SAC#sctp_assoc_change.assoc_id]),
      client_loop(S, Peer1, Port1, AssocId1, Peer2, Port2, SAC#sctp_assoc_change.assoc_id);
    {sctp, S, Peer1, Port1, Data} ->
      io:format("Association 1: received ~p~n", [Data]),
      client_loop(S, Peer1, Port1, AssocId1, Peer2, Port2, AssocId2);
    {sctp, S, Peer2, Port2, Data} ->
      io:format("Association 2: received ~p~n", [Data]),
      client_loop(S, Peer1, Port1, AssocId1, Peer2, Port2, AssocId2);
    Other ->
      io:format("Other ~p~n", [Other]),
      client_loop(S, Peer1, Port1, AssocId1, Peer2, Port2, AssocId2);
  after 5000 ->
    ok
  end.

See Also

gen_tcp(3), gen_udp(3), inet(3), RFC 2960 (Stream Control Transmission Protocol), Sockets API Extensions for SCTP
This module provides functions for communicating with sockets using the TCP/IP protocol.

The following code fragment is a simple example of a client connecting to a server at port 5678, transferring a binary, and closing the connection:

```erlang
client() ->
    SomeHostInNet = "localhost", % to make it runnable on one machine
    {ok, Sock} = gen_tcp:connect(SomeHostInNet, 5678,
                                  [binary, {packet, 0}]),
    ok = gen_tcp:send(Sock, "Some Data"),
    ok = gen_tcp:close(Sock).
```

At the other end, a server is listening on port 5678, accepts the connection, and receives the binary:

```erlang
server() ->
    {ok, LSock} = gen_tcp:listen(5678, [binary, {packet, 0},
                                         {active, false}]),
    {ok, Sock} = gen_tcp:accept(LSock),
    Bin = do_recv(Sock, []),
    ok = gen_tcp:close(Sock),
    ok = gen_tcp:close(LSock),
    Bin.

do_recv(Sock, Bs) ->
    case gen_tcp:recv(Sock, 0) of
        {ok, B} ->
            do_recv(Sock, [Bs, B]);
        {error, closed} ->
            {ok, list_to_binary(Bs)}
    end.
```

For more examples, see section Examples.

**Data Types**

```erlang
option() =
    [active, true | false | once | -32768..32767] |
    [buffer, integer() >= 0] |
    [delay_send, boolean()] |
    [deliver, port | term] |
    [dontroute, boolean()] |
    [exit_on_close, boolean()] |
    [header, integer() >= 0] |
    [high_msgq_watermark, integer() >= 1] |
    [high_watermark, integer() >= 0] |
    [keepalive, boolean()] |
    [linger, {boolean(), integer() >= 0}] |
    [low_msgq_watermark, integer() >= 0] |
    [low_watermark, integer() >= 0] |
    [mode, list | binary] |
    list | binary |
    [nodelay, boolean()] |
    [packet,
```

182 | Ericsson AB. All Rights Reserved.: Kernel
0 | 1 | 2 | 4 | raw | sunrm | asn1 | cdr | fcgi | line |
tpkt | http | http | http_bin | http_bin} |
{packet_size, integer() >= 0} |
{priority, integer() => 0} |
{raw, Protocol :: integer() => 0, OptionNum :: integer() => 0, ValueBin :: binary()} |
{recbuf, integer() => 0} |
{reuseaddr, boolean()} |
{send_timeout, integer() => 0 | infinity} |
{send_timeout_close, boolean()} |
{show_econnreset, boolean()} |
{sndbuf, integer() => 0} |
{tos, integer() => 0} |
{tclass, integer() => 0} |
{ttl, integer() => 0} |
{recvtos, boolean()} |
{recvvtclass, boolean()} |
{recvvttl, boolean()} |
{ipv6_v6only, boolean()}

pktoptions_value() = {pktoptions, inet:ancillary_data()}

If the platform implements the IPv4 option IP_PKTOPTIONS, or the IPv6 option IPV6_PKTOPTIONS or IPV6_2292PKTOPTIONS for the socket this value is returned from inet:getopts/2 when called with the option name pktoptions.

Note:

This option appears to be VERY Linux specific, and its existence in future Linux kernel versions is also worrying since the option is part of RFC 2292 which is since long (2003) obsoleted by RFC 3542 that explicitly removes this possibility to get packet information from a stream socket. For comparison: it has existed in FreeBSD but is now removed, at least since FreeBSD 10.
gen_tcp

\begin{verbatim}
inet:address_family() | 
{port, inet:port_number()} | 
{tcp_module, module()} | 
{netns, file:filename_all()} | 
{bind_to_device, binary()} | 
option()
listen_option() = 
{ip, inet:socket_address()} | 
{fd, Fd :: integer() >= 0} | 
{ifaddr, inet:socket_address()} | 
inet:address_family() | 
{port, inet:port_number()} | 
{backlog, B :: integer() >= 0} | 
{tcp_module, module()} | 
{netns, file:filename_all()} | 
{bind_to_device, binary()} | 
option()
socket()
\end{verbatim}

As returned by accept/1,2 and connect/3,4.

Exports

\textbf{accept}(ListenSocket) \rightarrow \{ok, Socket\} | \{error, Reason\}
\textbf{accept}(ListenSocket, Timeout) \rightarrow \{ok, Socket\} | \{error, Reason\}

Types:
\begin{itemize}
  \item \textbf{ListenSocket} = \textbf{socket()}
  \item \textbf{Timeout} = \textbf{timeout()}
  \item \textbf{Socket} = \textbf{socket()}
  \item \textbf{Reason} = \textbf{closed} | \textbf{timeout} | \textbf{system_limit} | \textbf{inet:posix()}
\end{itemize}

Accepts an incoming connection request on a listening socket. \textbf{Socket} must be a socket returned from \textbf{listen/2}. \textbf{Timeout} specifies a time-out value in milliseconds. Defaults to infinity.

Returns:
\begin{itemize}
  \item \{ok, Socket\} if a connection is established
  \item \{error, closed\} if \textbf{ListenSocket} is closed
  \item \{error, timeout\} if no connection is established within the specified time
  \item \{error, system_limit\} if all available ports in the Erlang emulator are in use
  \item A POSIX error value if something else goes wrong, see \textbf{inet(3)} for possible error values
\end{itemize}

Packets can be sent to the returned socket \textbf{Socket} using \textbf{send/2}. Packets sent from the peer are delivered as messages (unless \{active, false\} is specified in the option list for the listening socket, in which case packets are retrieved by calling \textbf{recv/2}):

\begin{verbatim}
{tcp, Socket, Data}
\end{verbatim}
Note:

The accept call does not have to be issued from the socket owner process. Using version 5.5.3 and higher of the emulator, multiple simultaneous accept calls can be issued from different processes, which allows for a pool of acceptor processes handling incoming connections.

close(Socket) -> ok
Types:
   Socket = socket()
Closes a TCP socket.

Note that in most implementations of TCP, doing a close does not guarantee that any data sent is delivered to the recipient before the close is detected at the remote side. If you want to guarantee delivery of the data to the recipient there are two common ways to achieve this.

• Use gen_tcp:shutdown(Socket, write) to signal that no more data is to be sent and wait for the read side of the socket to be closed.
• Use the socket option {packet, N} (or something similar) to make it possible for the receiver to close the connection when it knows it has received all the data.

connect(Address, Port, Options) -> {ok, Socket} | {error, Reason}
connect(Address, Port, Options, Timeout) ->
   {ok, Socket} | {error, Reason}
Types:
   Address = inet:socket_address() | inet:hostname()
   Port = inet:port_number()
   Options = [connect_option()]
   Timeout = timeout()
   Socket = socket()
   Reason = timeout | inet:posix()
Connects to a server on TCP port Port on the host with IP address Address. Argument Address can be a hostname or an IP address.

The following options are available:

{ip, Address}
   If the host has many network interfaces, this option specifies which one to use.

{ifaddr, Address}
   Same as {ip, Address}. If the host has many network interfaces, this option specifies which one to use.

{fd, integer() >= 0}
   If a socket has somehow been connected without using gen_tcp, use this option to pass the file descriptor for it. If {ip, Address} and/or {port, port_number()} is combined with this option, the fd is bound to the specified interface and port before connecting. If these options are not specified, it is assumed that the fd is already bound appropriately.

inet
   Sets up the socket for IPv4.
inet6
   Sets up the socket for IPv6.
local
   Sets up a Unix Domain Socket. See inet:local_address()
{port, Port}
   Specifies which local port number to use.
{tcp_module, module()}
   Overrides which callback module is used. Defaults to inet_tcp for IPv4 and inet6_tcp for IPv6.
Opt
   See inet:setopts/2.
Packets can be sent to the returned socket Socket using send/2. Packets sent from the peer are delivered as messages:
   {tcp, Socket, Data}
If the socket is in {active, N} mode (see inet:setopts/2 for details) and its message counter drops to 0, the following message is delivered to indicate that the socket has transitioned to passive({active, false}) mode:
   {tcp_passive, Socket}
If the socket is closed, the following message is delivered:
   {tcp_closed, Socket}
If an error occurs on the socket, the following message is delivered (unless {active, false} is specified in the option list for the socket, in which case packets are retrieved by calling recv/2):
   {tcp_error, Socket, Reason}
The optional Timeout parameter specifies a time-out in milliseconds. Defaults to infinity.

Note:
Keep in mind that if the underlying OS connect() call returns a timeout, gen_tcp:connect will also return a timeout (i.e. {error, etimedout}), even if a larger Timeout was specified.

Note:
The default values for options specified to connect can be affected by the Kernel configuration parameter inet_default_connect_options. For details, see inet(3).

controlling_process(Socket, Pid) -> ok | {error, Reason}
Types:
   Socket = socket()
   Pid = pid()
   Reason = closed | not_owner | badarg | inet:posix()
Assigns a new controlling process Pid to Socket. The controlling process is the process that receives messages from the socket. If called by any other process than the current controlling process, {error, not_owner} is returned.
If the process identified by Pid is not an existing local pid, \{error, badarg\} is returned. \{error, badarg\} may also be returned in some cases when Socket is closed during the execution of this function.

If the socket is set in active mode, this function will transfer any messages in the mailbox of the caller to the new controlling process. If any other process is interacting with the socket while the transfer is happening, the transfer may not work correctly and messages may remain in the caller's mailbox. For instance changing the sockets active mode before the transfer is complete may cause this.

\[\text{listen}(\text{Port}, \text{Options}) \rightarrow \{\text{ok, ListenSocket}\} | \{\text{error, Reason}\}\]

**Types:**
- \text{Port} = inet:port_number()
- \text{Options} = [listen_option()]
- \text{ListenSocket} = socket()
- \text{Reason} = system_limit | inet:posix()

Sets up a socket to listen on port \text{Port} on the local host.

If \text{Port} == 0, the underlying OS assigns an available port number, use inet:port/1 to retrieve it.

The following options are available:

**list**
- Received Packet is delivered as a list.

**binary**
- Received Packet is delivered as a binary.

\(\{\text{backlog, B}\}\)
- \text{B} is an integer \(\geq 0\). The backlog value defines the maximum length that the queue of pending connections can grow to. Defaults to 5.

\(\{\text{ip, Address}\}\)
- If the host has many network interfaces, this option specifies which one to listen on.

\(\{\text{port, Port}\}\)
- Specifies which local port number to use.

\(\{\text{fd, Fd}\}\)
- If a socket has somehow been connected without using \text{gen_tcp}, use this option to pass the file descriptor for it.

\(\{\text{ifaddr, Address}\}\)
- Same as \{ip, Address\}. If the host has many network interfaces, this option specifies which one to use.

**inet6**
- Sets up the socket for IPv6.

**inet**
- Sets up the socket for IPv4.

\(\{\text{tcp_module, module()}\}\)
- Overrides which callback module is used. Defaults to inet_tcp for IPv4 and inet6_tcp for IPv6.

\text{Opt}
- See \text{inet:setopts/2}.
The returned socket ListenSocket should be used in calls to accept/1,2 to accept incoming connection requests.

**Note:**

The default values for options specified to listen can be affected by the Kernel configuration parameter inet_default_listen_options. For details, see inet(3).

```
recv(Socket, Length) -> {ok, Packet} | {error, Reason}
recv(Socket, Length, Timeout) -> {ok, Packet} | {error, Reason}
```

**Types:**

- **Socket** = socket()
- **Length** = integer() \( \geq 0 \)
- **Timeout** = timeout()
- **Packet** = string() | binary() | HttpPacket
- **Reason** = closed | timeout | inet:posix()
- **HttpPacket** = term()

See the description of HttpPacket in erlang:decode_packet/3 in ERTS.

Receives a packet from a socket in passive mode. A closed socket is indicated by return value \{error, closed\}.

Argument **Length** is only meaningful when the socket is in raw mode and denotes the number of bytes to read.
If **Length** is 0, all available bytes are returned. If **Length** > 0, exactly **Length** bytes are returned, or an error; possibly discarding less than **Length** bytes of data when the socket is closed from the other side.

The optional **Timeout** parameter specifies a time-out in milliseconds. Defaults to infinity.

```
send(Socket, Packet) -> ok | {error, Reason}
```

**Types:**

- **Socket** = socket()
- **Packet** = iodata()
- **Reason** = closed | inet:posix()

Sends a packet on a socket.

There is no send call with a time-out option, use socket option send_timeout if time-outs are desired. See section Examples.

```
shutdown(Socket, How) -> ok | {error, Reason}
```

**Types:**

- **Socket** = socket()
- **How** = read | write | read_write
- **Reason** = inet:posix()

Closes a socket in one or two directions.

**How** == write means closing the socket for writing, reading from it is still possible.

If **How** == read or there is no outgoing data buffered in the **Socket** port, the socket is shut down immediately and any error encountered is returned in **Reason**.
If there is data buffered in the socket port, the attempt to shutdown the socket is postponed until that data is written to the kernel socket send buffer. If any errors are encountered, the socket is closed and \{error, closed\} is returned on the next \texttt{recv/2} or \texttt{send/2}.

Option \{exit_on_close, false\} is useful if the peer has done a shutdown on the write side.

**Examples**

The following example illustrates use of option \{active, once\} and multiple accepts by implementing a server as a number of worker processes doing accept on a single listening socket. Function \texttt{start/2} takes the number of worker processes and the port number on which to listen for incoming connections. If \texttt{LPort} is specified as 0, an ephemeral port number is used, which is why the start function returns the actual port number allocated:

\begin{Verbatim}
start(Num,LPort) ->
   case gen_tcp:listen(LPort,\{\{active, false\},\{packet,2\}\}) of
      \{ok, ListenSock\} ->
         start_servers(Num,ListenSock),
         \{ok, Port\} = inet:port(ListenSock),
         Port;
      \{error,Reason\} ->
         \{error,Reason\}
      end.

start_servers(0,_) ->
   ok;
start_servers(Num,LS) ->
   spawn(?MODULE,server,[LS]),
   start_servers(Num-1,LS).

server(LS) ->
   case gen_tcp:accept(LS) of
      \{ok,S\} ->
         loop(S),
         server(LS);
      Other ->
         io:format("accept returned ~w - goodbye!~-n",[Other]),
         ok
      end.

loop(S) ->
   inet:setopts(S,\{\{active,once\}\}),
   receive
      \{tcp,S,Data\} ->
         Answer = process(Data), % Not implemented in this example
         gen_tcp:send(S,Answer),
         loop(S);
      \{tcp_closed,S\} ->
         io:format("Socket \w closed [-w]-n",[S,self()]),
         ok
   end.
\end{Verbatim}

Example of a simple client:

\begin{Verbatim}
client(PortNo,Message) ->
   \{ok,Sock\} = gen_tcp:connect("localhost",PortNo,\{\{active,false\},
           \{packet,2\}\}),
   gen_tcp:send(Sock,Message),
   A = gen_tcp:recv(Sock,0),
   gen_tcp:close(Sock),
   A.
\end{Verbatim}

The \texttt{send} call does not accept a time-out option because time-outs on send is handled through socket option \texttt{send_timeout}. The behavior of a send operation with no receiver is mainly defined by the underlying TCP stack.
and the network infrastructure. To write code that handles a hanging receiver that can eventually cause the sender to hang on a send do like the following.

Consider a process that receives data from a client process to be forwarded to a server on the network. The process is connected to the server through TCP/IP and does not get any acknowledge for each message it sends, but has to rely on the send time-out option to detect that the other end is unresponsive. Option `send_timeout` can be used when connecting:

```erlang
{ok, Sock} = gen_tcp:connect(HostAddress, Port, [
  {active, false},
  {send_timeout, 5000},
  {packet, 2}]
),
loop(Sock), % See below
```

In the loop where requests are handled, send time-outs can now be detected:

```erlang
loop(Sock) ->
  receive
    {Client, send_data, Binary} ->
      case gen_tcp:send(Sock, [Binary]) of
        {error, timeout} ->
          io:format("Send timeout, closing!\n", []),
          handle_send_timeout(), % Not implemented here
          Client ! {self(), {error_sending, timeout}},
          %% Usually, it’s a good idea to give up in case of a
          %% send timeout, as you never know how much actually
          %% reached the server, maybe only a packet header?!
          gen_tcp:close(Sock);
        {error, OtherSendError} ->
          io:format("Some other error on socket (~p), closing", [
            OtherSendError]),
          Client ! {self(), {error_sending, OtherSendError}},
          gen_tcp:close(Sock);
        ok ->
          Client ! {self(), data_sent},
          loop(Sock)
      end.
  end.
```

Usually it suffices to detect time-outs on receive, as most protocols include some sort of acknowledge from the server, but if the protocol is strictly one way, option `send_timeout` comes in handy.
This module provides functions for communicating with sockets using the UDP protocol.

**Data Types**

```erlang
option() =
  {active, true | false | once | -32768..32767} |
  {add_membership, {inet:ip_address(), inet:ip_address()}} |
  {broadcast, boolean()} |
  {buffer, integer() >= 0} |
  {deliver, port | term} |
  {dontroute, boolean()} |
  {drop_membership, {inet:ip_address(), inet:ip_address()}} |
  {header, integer() >= 0} |
  {high_msgq_watermark, integer() >= 1} |
  {low_msgq_watermark, integer() >= 1} |
  {mode, list | binary} |
  {multicast_if, inet:ip_address()} |
  {multicast_loop, boolean()} |
  {multicast_ttl, integer() >= 0} |
  {priority, integer() >= 0} |
  {raw,
    Protocol :: integer() >= 0,
    OptionNum :: integer() >= 0,
    ValueBin :: binary()} |
  {read_packets, integer() >= 0} |
  {recbuf, integer() >= 0} |
  {reuseaddr, boolean()} |
  {sndbuf, integer() >= 0} |
  {tos, integer() >= 0} |
  {tclass, integer() >= 0} |
  {ttl, integer() >= 0} |
  {recvtos, boolean()} |
  {recv Herald class, boolean()} |
  {recvttl, boolean()} |
  {ipv6_v6only, boolean()}

option_name() =
  active | broadcast | buffer | deliver | dontroute | header | high_msgq_watermark | low_msgq_watermark | mode | multicast_if | multicast_loop | multicast_ttl | priority | 
  {raw,
    Protocol :: integer() >= 0,
    OptionNum :: integer() >= 0,
    ValueSpec ::
      (ValueSize :: integer() >= 0) | (ValueBin :: binary())} |
  read_packets | recbuf | reuseaddr | sndbuf | tos | tclass | 
  ttl | recvlos | recv Herald class | recv ttl | pktoptions |
ipv6_v6only
socket()
As returned by open/1,2.

Exports

close(Socket) -> ok
Types:
  Socket = socket()
Closes a UDP socket.

controlling_process(Socket, Pid) -> ok | {error, Reason}
Types:
  Socket = socket()
  Pid = pid()
  Reason = closed | not_owner | badarg | inet:posix()
Assigns a new controlling process Pid to Socket. The controlling process is the process that receives messages from the socket. If called by any other process than the current controlling process, {error, not_owner} is returned. If the process identified by Pid is not an existing local pid, {error, badarg} is returned. {error, badarg} may also be returned in some cases when Socket is closed during the execution of this function.

open(Port) -> {ok, Socket} | {error, Reason}
open(Port, Opts) -> {ok, Socket} | {error, Reason}
Types:
  Port = inet:port_number()
  Opts = [Option]
  Option =
    {ip, inet:socket_address()} | {fd, integer() >= 0} | {ifaddr, inet:socket_address()} | inet:address_family() |
    {port, inet:port_number()} | {netsns, file:filename_all()} | {bind_to_device, binary()} | option()
  Socket = socket()
  Reason = system_limit | inet:posix()
Associates a UDP port number (Port) with the calling process.
The following options are available:
list
  Received Packet is delivered as a list.
binary
  Received Packet is delivered as a binary.
(ip, Address)

If the host has many network interfaces, this option specifies which one to use.

(ifaddr, Address)

Same as (ip, Address). If the host has many network interfaces, this option specifies which one to use.

(fd, integer() >= 0)

If a socket has somehow been opened without using gen_udp, use this option to pass the file descriptor for it. If Port is not set to 0 and/or (ip, ip_address()) is combined with this option, the fd is bound to the specified interface and port after it is being opened. If these options are not specified, it is assumed that the fd is already bound appropriately.

inet6

Sets up the socket for IPv6.

inet

Sets up the socket for IPv4.

local

Sets up a Unix Domain Socket. See inet:local_address()

(udp_module, module())

Overrides which callback module is used. Defaults to inet_udp for IPv4 and inet6_udp for IPv6.

(multicast_if, Address)

Sets the local device for a multicast socket.

(multicast_loop, true | false)

When true, sent multicast packets are looped back to the local sockets.

(multicast_ttl, Integer)

Option multicast_ttl changes the time-to-live (TTL) for outgoing multicast datagrams to control the scope of the multicasts.

Datagrams with a TTL of 1 are not forwarded beyond the local network. Defaults to 1.

(add_membership, {MultiAddress, InterfaceAddress})

Joins a multicast group.

(drop_membership, {MultiAddress, InterfaceAddress})

Leaves a multicast group.

Opt

See inet:setopts/2.

The returned socket Socket is used to send packets from this port with send/4. When UDP packets arrive at the opened port, if the socket is in an active mode, the packets are delivered as messages to the controlling process:

{udp, Socket, IP, InPortNo, Packet} % Without ancillary data
{udp, Socket, IP, InPortNo, AncData, Packet} % With ancillary data

The message contains an AncData field if any of the socket options recvtos, recvtclass or recvttl are active, otherwise it does not.

If the socket is not in an active mode, data can be retrieved through the recv/2, 3 calls. Notice that arriving UDP packets that are longer than the receive buffer option specifies can be truncated without warning.
When a socket in \{active, N\} mode (see \texttt{inet:setopts/2} for details), transitions to passive \{active, false\}) mode, the controlling process is notified by a message of the following form:

\begin{verbatim}
{udp_passive, Socket}
\end{verbatim}

IP and InPortNo define the address from which Packet comes. Packet is a list of bytes if option list is specified. Packet is a binary if option binary is specified.

Default value for the receive buffer option is \{recbuf, 8192\}.

If Port \(= 0\), the underlying OS assigns a free UDP port, use \texttt{inet:port/1} to retrieve it.

\begin{verbatim}
recv(Socket, Length) -> {ok, RecvData} | {error, Reason}
recv(Socket, Length, Timeout) -> {ok, RecvData} | {error, Reason}
\end{verbatim}

Types:
\begin{verbatim}
Socket = socket()
Length = integer() \(\geq 0\)
Timeout = timeout()
RecvData =
  \{Address, Port, Packet\} | \{Address, Port, AncData, Packet\}
Address = inet:ip_address() | inet:returned_non_ip_address()
Port = inet:port_number()
AncData = inet:ancillary_data()
Packet = string() | binary()
Reason = not_owner | timeout | inet:posix()
\end{verbatim}

Receives a packet from a socket in passive mode. Optional parameter Timeout specifies a time-out in milliseconds. Defaults to infinity.

If any of the socket options recvtos, recvtclass or recvttl are active, the RecvData tuple contains an AncData field, otherwise it does not.

\begin{verbatim}
send(Socket, Destination, Packet) -> ok | {error, Reason}
send(Socket, Host, Port, Packet) -> ok | {error, Reason}
\end{verbatim}

Types:
\begin{verbatim}
Socket = socket()
Destination =
  \{inet:ip_address(), inet:port_number()\} | inet:family_address()
Packet = iodata()
Reason = not_owner | inet:posix()
\end{verbatim}

Sends a packet to the specified Destination.

This function is equivalent to \texttt{send(Socket, Destination, [], Packet)}.
Sends a packet to the specified Host and Port.

This clause is equivalent to `send(Socket, Host, Port, [], Packet).

```
send(Socket, Destination, AncData, Packet) -> ok | {error, Reason}
```

**Types:**
- `Socket = socket()`
- `Destination = {inet:ip_address(), inet:port_number()} | inet:family_address()`
- `AncData = inet:ancillary_data()`
- `Packet = iodata()`
- `Reason = not_owner | inet:posix()`

Sends a packet to the specified Destination with ancillary data AncData.

**Note:**
The ancillary data AncData contains options that for this single message override the default options for the socket, an operation that may not be supported on all platforms, and if so return `{error, einval}`. Using more than one of an ancillary data item type may also not be supported. AncData =:= [] is always supported.

```
send(Socket, Destination, PortZero, Packet) -> ok | {error, Reason}
```

**Types:**
- `Socket = socket()`
- `Destination = {inet:ip_address(), inet:port_number()} | inet:family_address()`
- `PortZero = inet:port_number()`
- `Packet = iodata()`
- `Reason = not_owner | inet:posix()`

Sends a packet to the specified Destination. Since Destination is complete, PortZero is redundant and has to be 0.

This is a legacy clause mostly for `Destination = {local, Binary}` where PortZero is superfluous. It is equivalent to `send(Socket, Destination, [], Packet), the clause right above here.

```
send(Socket, Host, Port, AncData, Packet) -> ok | {error, Reason}
```

**Types:**
- `Socket = socket()`
- `Host =`
Sends a packet to the specified Host and Port, with ancillary data AncData.

Argument Host can be a hostname or a socket address, and Port can be a port number or a service name atom. These are resolved into a Destination and after that this function is equivalent to send(Socket, Destination, AncData, Packet), read there about ancillary data.
This module consists of the following services:

- Registration of global names
- Global locks
- Maintenance of the fully connected network

These services are controlled through the process `global_name_server` that exists on every node. The global name server starts automatically when a node is started. With the term `global` is meant over a system consisting of many Erlang nodes.

The ability to globally register names is a central concept in the programming of distributed Erlang systems. In this module, the equivalent of the `register/2` and `whereis/1` BIFs (for local name registration) are provided, but for a network of Erlang nodes. A registered name is an alias for a process identifier (pid). The global name server monitors globally registered pids. If a process terminates, the name is also globally unregistered.

The registered names are stored in replica global name tables on every node. There is no central storage point. Thus, the translation of a name to a pid is fast, as it is always done locally. For any action resulting in a change to the global name table, all tables on other nodes are automatically updated.

Global locks have lock identities and are set on a specific resource. For example, the specified resource can be a pid. When a global lock is set, access to the locked resource is denied for all resources other than the lock requester.

Both the registration and lock services are atomic. All nodes involved in these actions have the same view of the information.

The global name server also performs the critical task of continuously monitoring changes in node configuration. If a node that runs a globally registered process goes down, the name is globally unregistered. To this end, the global name server subscribes to `nodeup` and `nodedown` messages sent from module `net_kernel`. Relevant Kernel application variables in this context are `net_setuptime`, `net_ticktime`, and `dist_auto_connect`. See also `kernel(6)`.

The name server also maintains a fully connected network. For example, if node `N1` connects to node `N2` (which is already connected to `N3`), the global name servers on the nodes `N1` and `N3` ensure that also `N1` and `N3` are connected. If this is not desired, command-line flag `-connect_all false` can be used (see also `erl(1)`). In this case, the name registration service cannot be used, but the lock mechanism still works.

If the global name server fails to connect nodes (`N1` and `N3` in the example), a warning event is sent to the error logger. The presence of such an event does not exclude the nodes to connect later (you can, for example, try command `rpc:call(N1, net_adm, ping, [N2])` in the Erlang shell), but it indicates a network problem.

**Note:**

If the fully connected network is not set up properly, try first to increase the value of `net_setuptime`. 

Data Types

\[ id() = \{\text{ResourceId} : \text{term()}, \text{LockRequesterId} : \text{term()}\} \]

Exports

\[ \text{del_lock}(Id) \rightarrow \text{true} \]
\[ \text{del_lock}(Id, \text{Nodes}) \rightarrow \text{true} \]

Types:
- \( Id = id() \)
- \( \text{Nodes} = [\text{node()}] \)

Deletes the lock \( Id \) synchronously.

\[ \text{notify_all_name}(\text{Name}, \text{Pid}1, \text{Pid}2) \rightarrow \text{none} \]

Types:
- \( \text{Name} = \text{term()} \)
- \( \text{Pid}1 = \text{Pid}2 = \text{pid()} \)

Can be used as a name resolving function for \( \text{register_name}/3 \) and \( \text{re_register_name}/3 \).

The function unregisters both pids and sends the message \( \{\text{global_name_conflict}, \text{Name}, \text{OtherPid}\} \)

to both processes.

\[ \text{random_exit_name}(\text{Name}, \text{Pid}1, \text{Pid}2) \rightarrow \text{pid()} \]

Types:
- \( \text{Name} = \text{term()} \)
- \( \text{Pid}1 = \text{Pid}2 = \text{pid()} \)

Can be used as a name resolving function for \( \text{register_name}/3 \) and \( \text{re_register_name}/3 \).

The function randomly selects one of the pids for registration and kills the other one.

\[ \text{random_notify_name}(\text{Name}, \text{Pid}1, \text{Pid}2) \rightarrow \text{pid()} \]

Types:
- \( \text{Name} = \text{term()} \)
- \( \text{Pid}1 = \text{Pid}2 = \text{pid()} \)

Can be used as a name resolving function for \( \text{register_name}/3 \) and \( \text{re_register_name}/3 \).

The function randomly selects one of the pids for registration, and sends the message \( \{\text{global_name_conflict}, \text{Name}\} \)
to the other pid.

\[ \text{re_register_name}(\text{Name}, \text{Pid}) \rightarrow \text{yes} \]
\[ \text{re_register_name}(\text{Name}, \text{Pid}, \text{Resolve}) \rightarrow \text{yes} \]

Types:
- \( \text{Name} = \text{term()} \)
- \( \text{Pid} = \text{pid()} \)
- \( \text{Resolve} = \text{method()} \)
- \( \text{method()} = \)
fun((Name :: term(), Pid :: pid(), Pid2 :: pid()) ->
    pid() | none)

[Module, Function] is also allowed.

Atomically changes the registered name Name on all nodes to refer to Pid.

Function Resolve has the same behavior as in register_name/2,3.

register_name(Name, Pid) -> yes | no
register_name(Name, Pid, Resolve) -> yes | no

Types:
Name = term()
Pid = pid()
Resolve = method()
method() =
    fun((Name :: term(), Pid :: pid(), Pid2 :: pid()) ->
        pid() | none)

[Module, Function] is also allowed for backward compatibility, but its use is deprecated.

Globally associates name Name with a pid, that is, globally notifies all nodes of a new global name in a network of Erlang nodes.

When new nodes are added to the network, they are informed of the globally registered names that already exist. The network is also informed of any global names in newly connected nodes. If any name clashes are discovered, function Resolve is called. Its purpose is to decide which pid is correct. If the function crashes, or returns anything other than one of the pids, the name is unregistered. This function is called once for each name clash.

Warning:
If you plan to change code without restarting your system, you must use an external fun (fun Module:Function/Arity) as function Resolve. If you use a local fun, you can never replace the code for the module that the fun belongs to.

Three predefined resolve functions exist: random_exit_name/3, random_notify_name/3, and notify_all_name/3. If no Resolve function is defined, random_exit_name is used. This means that one of the two registered processes is selected as correct while the other is killed.

This function is completely synchronous, that is, when this function returns, the name is either registered on all nodes or none.

The function returns yes if successful, no if it fails. For example, no is returned if an attempt is made to register an already registered process or to register a process with a name that is already in use.

Note:
Releases up to and including Erlang/OTP R10 did not check if the process was already registered. The global name table could therefore become inconsistent. The old (buggy) behavior can be chosen by giving the Kernel application variable global_multi_name_action the value allow.

If a process with a registered name dies, or the node goes down, the name is unregistered on all nodes.

registered_names() -> [Name]

Types:
Name = term()

Returns a list of all globally registered names.

send(Name, Msg) -> Pid

Types:
   Name = Msg = term()
   Pid = pid()

Sends message Msg to the pid globally registered as Name.
If Name is not a globally registered name, the calling function exits with reason {badarg, {Name, Msg}}.

set_lock(Id) -> boolean()
set_lock(Id, Nodes) -> boolean()
set_lock(Id, Nodes, Retries) -> boolean()

Types:
   Id = id()
   Nodes = [node()]
   Retries = retries()
   id() = {ResourceId :: term(), LockRequesterId :: term()}
   retries() = integer() >= 0 | infinity

Sets a lock on the specified nodes (or on all nodes if none are specified) on ResourceId for LockRequesterId.
If a lock already exists on ResourceId for another requester than LockRequesterId, and Retries is not equal
to 0, the process sleeps for a while and tries to execute the action later. When Retries attempts have been made,
false is returned, otherwise true. If Retries is infinity, true is eventually returned (unless the lock is
never released).
If no value for Retries is specified, infinity is used.
This function is completely synchronous.
If a process that holds a lock dies, or the node goes down, the locks held by the process are deleted.
The global name server keeps track of all processes sharing the same lock, that is, if two processes set the same lock,
both processes must delete the lock.
This function does not address the problem of a deadlock. A deadlock can never occur as long as processes only lock
one resource at a time. A deadlock can occur if some processes try to lock two or more resources. It is up to the
application to detect and rectify a deadlock.

Note:

Avoid the following values of ResourceId, otherwise Erlang/OTP does not work properly:
- dist_ac
- global
- mnesia_adjust_logWrites
- mnesia_table_lock
- pg2
sync() -> ok | {error, Reason :: term()}

Synchronizes the global name server with all nodes known to this node. These are the nodes that are returned from `erlang:nodes()`. When this function returns, the global name server receives global information from all nodes. This function can be called when new nodes are added to the network.

The only possible error reason `Reason` is "global_groups definition error", `Error`.

trans(Id, Fun) -> Res | aborted
trans(Id, Fun, Nodes) -> Res | aborted
trans(Id, Fun, Nodes, Retries) -> Res | aborted

Types:

- `Id` = `id()`
- `Fun` = `trans_fun()`
- `Nodes` = `[node()]`
- `Retries` = `retries()`
- `Res` = `term()`
- `retries()` = `integer() >= 0 | infinity`
- `trans_fun()` = `function() | {module(), atom()}`

Sets a lock on `Id` (using `set_lock/3`). If this succeeds, `Fun()` is evaluated and the result `Res` is returned. Returns `aborted` if the lock attempt fails. If `Retries` is set to `infinity`, the transaction does not abort.

`infinity` is the default setting and is used if no value is specified for `Retries`.

unregister_name(Name) -> term()

Types:

- `Name` = `term()`

Removes the globally registered name `Name` from the network of Erlang nodes.

whereis_name(Name) -> pid() | undefined

Types:

- `Name` = `term()`

Returns the pid with the globally registered name `Name`. Returns `undefined` if the name is not globally registered.

See Also

global_group(3), net_kernel(3)
This module makes it possible to partition the nodes of a system into global groups. Each global group has its own global namespace, see global(3).

The main advantage of dividing systems into global groups is that the background load decreases while the number of nodes to be updated is reduced when manipulating globally registered names.

The Kernel configuration parameter global_groups defines the global groups (see also kernel(6) and config(4)):

```
{global_groups, [GroupTuple :: group_tuple()]}[
```

For the processes and nodes to run smoothly using the global group functionality, the following criteria must be met:

- An instance of the global group server, global_group, must be running on each node. The processes are automatically started and synchronized when a node is started.
- All involved nodes must agree on the global group definition, otherwise the behavior of the system is undefined.
- All nodes in the system must belong to exactly one global group.

In the following descriptions, a group node is a node belonging to the same global group as the local node.

**Data Types**

- `group_tuple() = {GroupName :: group_name(), [node()]} | {GroupName :: group_name(), PublishType :: publish_type(), [node()]}`

A GroupTuple without PublishType is the same as a GroupTuple with PublishType equal to normal.

- `group_name() = atom()`
- `publish_type() = hidden | normal`

A node started with command-line flag `-hidden` (see erl(1)) is said to be a hidden node. A hidden node establishes hidden connections to nodes not part of the same global group, but normal (visible) connections to nodes part of the same global group.

A global group defined with PublishType equal to hidden is said to be a hidden global group. All nodes in a hidden global group are hidden nodes, whether they are started with command-line flag `-hidden` or not.

- `name() = atom()` A registered name.
- `where() = {node, node()} | {group, group_name()}`

**Exports**

```
global_groups() -> {GroupName, GroupNames} | undefined
```

Types:
GroupName = group_name()
GroupNames = [GroupName]

Returns a tuple containing the name of the global group that the local node belongs to, and the list of all other known group names. Returns undefined if no global groups are defined.

info() -> [info_item()]

Types:

info_item() =
  {state, State :: sync_state()} |
  {own_group_name, GroupName :: group_name()} |
  {own_group_nodes, Nodes :: [node()]} |
  {synced_nodes, Nodes :: [node()]} |
  {sync_error, Nodes :: [node()]} |
  {no_contact, Nodes :: [node()]} |
  {other_groups, Groups :: [group_tuple()]} |
  {monitoring, Pids :: [pid()]} |

sync_state() = no_conf | synced

Returns a list containing information about the global groups. Each list element is a tuple. The order of the tuples is undefined.

{state, State}

If the local node is part of a global group, State is equal to synced. If no global groups are defined, State is equal to no_conf.

{own_group_name, GroupName}
The name (atom) of the group that the local node belongs to.

{own_group_nodes, Nodes}
A list of node names (atoms), the group nodes.

{synced_nodes, Nodes}
A list of node names, the group nodes currently synchronized with the local node.

{sync_error, Nodes}
A list of node names, the group nodes with which the local node has failed to synchronize.

{no_contact, Nodes}
A list of node names, the group nodes to which there are currently no connections.

{other_groups, Groups}
Groups is a list of tuples {GroupName, Nodes}, specifying the name and nodes of the other global groups.

{monitoring, Pids}
A list of pids, specifying the processes that have subscribed to nodeup and nodedown messages.

monitor_nodes(Flag) -> ok

Types:

Flag = boolean()

Depending on Flag, the calling process starts subscribing (Flag equal to true) or stops subscribing (Flag equal to false) to node status change messages.
A process that has subscribed receives the messages \{nodeup, \text{Node}\} and \{nodedown, \text{Node}\} when a group node connects or disconnects, respectively.

\textbf{own\_nodes()} \rightarrow \textbf{Nodes}

\textbf{Types:}

Nodes = [\text{Node} :: \text{node()}]

Returns the names of all group nodes, regardless of their current status.

\textbf{registered\_names(Where)} \rightarrow \textbf{Names}

\textbf{Types:}

Where = \text{where()}

Names = [\text{Name} :: \text{name()}}

Returns a list of all names that are globally registered on the specified node or in the specified global group.

\textbf{send(Name, Msg)} \rightarrow \text{pid()} | \{\text{badarg, \{Name, Msg\}}}

\textbf{send(Where, Name, Msg)} \rightarrow \text{pid()} | \{\text{badarg, \{Name, Msg\}}}

\textbf{Types:}

Where = \text{where()}

Name = \text{name()}

Msg = \text{term()}

Searches for Name, globally registered on the specified node or in the specified global group, or (if argument \text{Where} is not provided) in any global group. The global groups are searched in the order that they appear in the value of configuration parameter \text{global\_groups}.

If Name is found, message \text{Msg} is sent to the corresponding \text{pid}. The \text{pid} is also the return value of the function. If the name is not found, the function returns \{\text{badarg, \{Name, Msg\}}\}.

\textbf{sync()} \rightarrow \text{ok}

Synchronizes the group nodes, that is, the global name servers on the group nodes. Also checks the names globally registered in the current global group and unregisters them on any known node not part of the group.

If synchronization is not possible, an error report is sent to the error logger (see also \text{error\_logger(3)}).

Returns \{\text{error, \{'invalid global\_groups definition', Bad}\}} if configuration parameter \text{global\_groups} has an invalid value \text{Bad}.

\textbf{whereis\_name(Name)} \rightarrow \text{pid()} | \text{undefined}

\textbf{whereis\_name(Where, Name)} \rightarrow \text{pid()} | \text{undefined}

\textbf{Types:}

Where = \text{where()}

Name = \text{name()}

Searches for Name, globally registered on the specified node or in the specified global group, or (if argument \text{Where} is not provided) in any global group. The global groups are searched in the order that they appear in the value of configuration parameter \text{global\_groups}.

If Name is found, the corresponding \text{pid} is returned. If the name is not found, the function returns \text{undefined}. 

204 | Ericsson AB. All Rights Reserved.: Kernel
Notes

- In the situation where a node has lost its connections to other nodes in its global group, but has connections to nodes in other global groups, a request from another global group can produce an incorrect or misleading result. For example, the isolated node can have inaccurate information about registered names in its global group.
- Function `send/2,3` is not secure.
- Distribution of applications is highly dependent of the global group definitions. It is not recommended that an application is distributed over many global groups, as the registered names can be moved to another global group at failover/takeover. Nothing prevents this to be done, but the application code must then handle the situation.

See Also

global(3), erl(1)
heart

Erlang module

This module contains the interface to the heart process. heart sends periodic heartbeats to an external port program, which is also named heart. The purpose of the heart port program is to check that the Erlang runtime system it is supervising is still running. If the port program has not received any heartbeats within HEART_BEAT_TIMEOUT seconds (defaults to 60 seconds), the system can be rebooted.

An Erlang runtime system to be monitored by a heart program is to be started with command-line flag -heart (see also erl(1)). The heart process is then started automatically:

```
% erl -heart ...
```

If the system is to be rebooted because of missing heartbeats, or a terminated Erlang runtime system, environment variable HEART_COMMAND must be set before the system is started. If this variable is not set, a warning text is printed but the system does not reboot.

To reboot on Windows, HEART_COMMAND can be set to heart -shutdown (included in the Erlang delivery) or to any other suitable program that can activate a reboot.

The environment variable HEART_BEAT_TIMEOUT can be used to configure the heart time-outs; it can be set in the operating system shell before Erlang is started or be specified at the command line:

```
% erl -heart -env HEART_BEAT_TIMEOUT 30 ...
```

The value (in seconds) must be in the range 10 < X <= 65535.

When running on OSs lacking support for monotonic time, heart is susceptible to system clock adjustments of more than HEART_BEAT_TIMEOUT seconds. When this happens, heart times out and tries to reboot the system. This can occur, for example, if the system clock is adjusted automatically by use of the Network Time Protocol (NTP).

If a crash occurs, an erl_crash.dump is not written unless environment variable ERL_CRASH_DUMP_SECONDS is set:

```
% erl -heart -env ERL_CRASH_DUMP_SECONDS 10 ...
```

If a regular core dump is wanted, let heart know by setting the kill signal to abort using environment variable HEART_KILL_SIGNAL=SIGABRT. If unset, or not set to SIGABRT, the default behavior is a kill signal using SIGKILL:

```
% erl -heart -env HEART_KILL_SIGNAL SIGABRT ...
```

If heart should not kill the Erlang runtime system, this can be indicated using the environment variable HEART_NO_KILL=TRUE. This can be useful if the command executed by heart takes care of this, for example as part of a specific cleanup sequence. If unset, or not set to TRUE, the default behaviour will be to kill as described above.

```
% erl -heart -env HEART_NO_KILL 1 ...
```

Furthermore, ERL_CRASH_DUMP_SECONDS has the following behavior on heart:
ERL_CRASH_DUMP_SECONDS=0
Suppresses the writing of a crash dump file entirely, thus rebooting the runtime system immediately. This is the same as not setting the environment variable.

ERL_CRASH_DUMP_SECONDS=-1
Setting the environment variable to a negative value does not reboot the runtime system until the crash dump file is completely written.

ERL_CRASH_DUMP_SECONDS=S
heart waits for S seconds to let the crash dump file be written. After S seconds, heart reboots the runtime system, whether the crash dump file is written or not.

In the following descriptions, all functions fail with reason badarg if heart is not started.

Data Types
heart_option() = check_schedulers

Exports

set_cmd(Cmd) -> ok | {error, {bad_cmd, Cmd}}
Types:
    Cmd = string()
Sets a temporary reboot command. This command is used if a HEART_COMMAND other than the one specified with the environment variable is to be used to reboot the system. The new Erlang runtime system uses (if it misbehaves) environment variable HEART_COMMAND to reboot.

Limitations: Command string Cmd is sent to the heart program as an ISO Latin-1 or UTF-8 encoded binary, depending on the filename encoding mode of the emulator (see file:native_name_encoding/0). The size of the encoded binary must be less than 2047 bytes.

clear_cmd() -> ok
Clears the temporary boot command. If the system terminates, the normal HEART_COMMAND is used to reboot.

get_cmd() -> {ok, Cmd}
Types:
    Cmd = string()
Gets the temporary reboot command. If the command is cleared, the empty string is returned.

set_callback(Module, Function) ->
    ok | {error, {bad_callback, {Module, Function}}}
Types:
    Module = Function = atom()
This validation callback will be executed before any heartbeat is sent to the port program. For the validation to succeed it needs to return with the value ok.
An exception within the callback will be treated as a validation failure.
The callback will be removed if the system reboots.
clear_callback() -> ok
Removes the validation callback call before heartbeats.

get_callback() -> {ok, {Module, Function}} | none
Types:
  Module = Function = atom()
Get the validation callback. If the callback is cleared, none will be returned.

set_options(Options) -> ok | {error, {bad_options, Options}}
Types:
  Options = [heart_option()]
Valid options set_options are:
  check_schedulers
    If enabled, a signal will be sent to each scheduler to check its responsiveness. The system check occurs before any heartbeat sent to the port program. If any scheduler is not responsive enough the heart program will not receive its heartbeat and thus eventually terminate the node.
Returns with the value ok if the options are valid.

get_options() -> {ok, Options} | none
Types:
  Options = [atom()]
Returns {ok, Options} where Options is a list of current options enabled for heart. If the callback is cleared, none will be returned.
This module provides access to TCP/IP protocols.

See also ERTS User’s Guide: Inet Configuration for more information about how to configure an Erlang runtime system for IP communication.

The following two Kernel configuration parameters affect the behavior of all sockets opened on an Erlang node:

• `inet_default_connect_options` can contain a list of default options used for all sockets returned when doing `connect`.

• `inet_default_listen_options` can contain a list of default options used when issuing a `listen` call.

When `accept` is issued, the values of the listening socket options are inherited. No such application variable is therefore needed for `accept`.

Using the Kernel configuration parameters above, one can set default options for all TCP sockets on a node, but use this with care. Options such as `{delay_send, true}` can be specified in this way. The following is an example of starting an Erlang node with all sockets using delayed send:

```
$ erl -sname test -kernel \
inet_default_connect_options '[]\ninet_default_listen_options '[]
```

Notice that default option `{active, true}` cannot be changed, for internal reasons.

Addresses as inputs to functions can be either a string or a tuple. For example, the IP address 150.236.20.73 can be passed to `gethostbyaddr/1`, either as string "150.236.20.73" or as tuple `{150, 236, 20, 73}.

### IPv4 address examples:

<table>
<thead>
<tr>
<th>Address</th>
<th>ip_address()</th>
</tr>
</thead>
<tbody>
<tr>
<td>127.0.0.1</td>
<td><code>{127,0,0,1}</code></td>
</tr>
<tr>
<td>192.168.42.2</td>
<td><code>{192,168,42,2}</code></td>
</tr>
</tbody>
</table>

### IPv6 address examples:

<table>
<thead>
<tr>
<th>Address</th>
<th>ip_address()</th>
</tr>
</thead>
<tbody>
<tr>
<td>::</td>
<td><code>{0,0,0,0,0,0,0,1}</code></td>
</tr>
<tr>
<td>::192.168.42.2</td>
<td><code>{0,0,0,0,0,0,192 bsl 8} bor 168,(42 bsl 8) bor 2}</code></td>
</tr>
<tr>
<td>::FFFF:192.168.42.2</td>
<td><code>{0,0,0,0,0,16#FFF,(192 bsl 8) bor 168,(42 bsl 8) bor 2}</code></td>
</tr>
<tr>
<td>3ffe:b80:1f8d:2:204:acff:fe17:bf38</td>
<td><code>{16#3ffe,16#b80,16#1f8d,16#2,16#204,16#acff,16#fe17,16#bf38}</code></td>
</tr>
<tr>
<td>fe80::204:acff:fe17:bf38</td>
<td><code>{16#fe80,0,0,0,0,16#204,16#acff,16#fe17,16#bf38}</code></td>
</tr>
</tbody>
</table>

Function `parse_address/1` can be useful:

```
1> inet:parse_address("192.168.42.2").
{ok,\{192,168,42,2\}}
2> inet:parse_address("::FFFF:192.168.42.2").
{ok,\{0,0,0,0,0,16#55535,49328,18754\}}
```
**Data Types**

hostent() =
   #hostent{h_name = inet:hostname(),
        h_aliases = [inet:hostname()],
        h_addrtype = inet | inet6,
        h_length = integer() >= 0,
        h_addr_list = [inet:ip_address()]

The record is defined in the Kernel include file "inet.hrl".

Add the following directive to the module:

   -include_lib("kernel/include/inet.hrl").

hostname() = atom() | string()
ip_address() = ip4_address() | ip6_address()
ip4_address() = {0..255, 0..255, 0..255, 0..255}
ip6_address() =
   {0..65535, 0..65535, 0..65535, 0..65535, 0..65535, 0..65535, 0..65535, 0..65535}
port_number() = 0..65535
family_address() =
   inet_address() | inet6_address() | local_address()

A general address format on the form \((\text{Family, Destination})\) where Family is an atom such as local and the format of Destination depends on Family, and is a complete address (for example an IP address including port number).

inet_address() =
   {inet, {ip4_address() | any | loopback, port_number()}}

**Warning:**

This address format is for now experimental and for completeness to make all address families have a \((\text{Family, Destination})\) representation.

inet6_address() =
   {inet6, {ip6_address() | any | loopback, port_number()}}

**Warning:**

This address format is for now experimental and for completeness to make all address families have a \((\text{Family, Destination})\) representation.

local_address() = {local, File :: binary() | string()}

This address family only works on Unix-like systems.
file is normally a file pathname in a local filesystem. It is limited in length by the operating system, traditionally to 108 bytes.

A binary() is passed as is to the operating system, but a string() is encoded according to the system filename encoding mode.

Other addresses are possible, for example Linux implements "Abstract Addresses". See the documentation for Unix Domain Sockets on your system, normally unix in manual section 7.

In most API functions where you can use this address family the port number must be 0.

socket_address() =
  ip_address() | any | loopback | local_address()
socket_getopt() =
  gen_sctp:option_name() |
  gen_tcp:option_name() |
  gen_udp:option_name()
socket_setopt() =
  gen_sctp:option() | gen_tcp:option() | gen_udp:option()
returned_non_ip_address() =
  {local, binary()} | {unspec, <<>>} | {undefined, any()}

Addresses besides ip_address() ones that are returned from socket API functions. See in particular local_address(). The unspec family corresponds to AF_UNSPEC and can occur if the other side has no socket address. The undefined family can only occur in the unlikely event of an address family that the VM does not recognize.

ancillary_data() =
  [{tos, byte()} | {tclass, byte()} | {ttl, byte()}]
Ancillary data received with the data packet, read with the socket option pktoptions from a TCP socket, or to set in a call to gen_udp:send/4 or gen_udp:send/5.

The value(s) correspond to the currently active socket options recvtos, recvtclass and recvttl, or for a single send operation the option(s) to override the currently active socket option(s).

getifaddrs_ifopts() =
  [Ifopt ::
    {flags,
      Flags ::
        [up | broadcast | loopback | pointtopoint |
          running | multicast]} |
    {addr, Addr :: ip_address()} |
    {netmask, Netmask :: ip_address()} |
    {broadaddr, Broadaddr :: ip_address()} |
    {dstaddr, Dstaddr :: ip_address()} |
    {hwaddr, Hwaddr :: [byte()]}
  ]
Interface address description list returned from getifaddrs/0,1 for a named interface, translated from the returned data of the POSIX API function getaddrinfo().

Hwaddr is hardware dependent, for example, on Ethernet interfaces it is the 6-byte Ethernet address (MAC address (EUI-48 address)).

The tuples {addr,Addr}, {netmask,Netmask}, and possibly {broadaddr,Broadaddr} or {dstaddr,Dstaddr} are repeated in the list if the interface has got multiple addresses. An interface may have multiple {flag,=} tuples for example if it has different flags for different address families. Multiple
\{(hwaddr, Hwaddr)\} tuples is hard to say anything definite about, though. The tuple \{(flag, Flags)\} is mandatory, all others are optional.

Do not rely too much on the order of Flags atoms or the Ifopt tuples. There are however some rules:

- A \{flag, _\} tuple applies to all other tuples that follow.
- Immediately after \{addr, _\} follows \{netmask, _\}.
- Immediately thereafter may \{broadaddr, _\} follow if broadcast is member of Flags, or \{dstaddr, _\} if pointtopoint is member of Flags. Both \{dstaddr, _\} and \{broadaddr, _\} does not occur for the same \{addr, _\}.
- Any \{netmask, _\}, \{broadaddr, _\}, or \{dstaddr, _\} tuples that follow an \{addr, Addr\} tuple concerns the address Addr.

The tuple \{hwaddr, _\} is not returned on Solaris, as the hardware address historically belongs to the link layer and it is not returned by the Solaris API function getaddrsinfo().

\begin{Verbatim}
Warning:

On Windows, the data is fetched from different OS API functions, so the Netmask and Broadaddr values may be calculated, just as some Flags values.
\end{Verbatim}

\texttt{posix()} =
\begin{verbatim}
eaddrinuse | eaddrnotavail | eafnosupport | ealready |
econnaborted | econnrefused | econnreset | edestaddrreq |
ehostdown | ehostunreach | einprogress | eisconn | emsgsize |
enetdown | enetunreach | enopkg | enoprotoopt | enotconn |
enotty | enotsock | eproto | eprotonosupport | eprototype |
esocktnosupport | etimedout | ewouldblock | exbadport |
exbadseq |
\end{verbatim}

An atom that is named from the POSIX error codes used in Unix, and in the runtime libraries of most C compilers. See section POSIX Error Codes.

\texttt{socket()}

See \texttt{gen_tcp:type-socket} and \texttt{gen_udp:type-socket}.

\texttt{address_family()} = \texttt{inet | inet6 | local}
\texttt{socket_protocol()} = \texttt{tcp | udp | sctp}
\texttt{stat_option()} =
\begin{verbatim}
recv_cnt | recv_max | recv_avg | recv_oct | recv_dvi |
send_cnt | send_max | send_avg | send_oct | send_pend
\end{verbatim}

Exports

\texttt{close(Socket) -> ok}
Types:
\begin{verbatim}
Socket = socket()
\end{verbatim}
Closes a socket of any type.

\texttt{format_error(Reason) -> string()}
Types:
Reason = posix() | system_limit

Returns a diagnostic error string. For possible POSIX values and corresponding strings, see section POSIX Error Codes.

get_rc() -> [{Par :: atom(), Val :: any()} | {Par :: atom(), Val1 :: any(), Val2 :: any()}]

Returns the state of the Inet configuration database in form of a list of recorded configuration parameters. For more information, see ERTS User's Guide: Inet Configuration.

Only actual parameters with other than default values are returned, for example not directives that specify other sources for configuration parameters nor directives that clear parameters.

getaddr(Host, Family) -> {ok, Address} | {error, posix()}

Types:
  Host = ip_address() | hostname()
  Family = address_family()
  Address = ip_address()

Returns the IP address for Host as a tuple of integers. Host can be an IP address, a single hostname, or a fully qualified hostname.

getaddrs(Host, Family) -> {ok, Addresses} | {error, posix()}

Types:
  Host = ip_address() | hostname()
  Family = address_family()
  Addresses = [ip_address()]

Returns a list of all IP addresses for Host. Host can be an IP address, a single hostname, or a fully qualified hostname.

gethostbyaddr(Address) -> {ok, Hostent} | {error, posix()}

Types:
  Address = string() | ip_address()
  Hostent = hostent()

Returns a hostent record for the host with the specified address.

gethostbyname(Hostname) -> {ok, Hostent} | {error, posix()}

Types:
  Hostname = hostname()
  Hostent = hostent()

Returns a hostent record for the host with the specified hostname.

If resolver option inet6 is true, an IPv6 address is looked up.

gethostbyname(Hostname, Family) -> {ok, Hostent} | {error, posix()}

Types:
Hostname = hostname()
Family = address_family()
Hostent = hostent()

Returns a hostent record for the host with the specified name, restricted to the specified address family.

gethostname() -> {ok, Hostname}

Types:
  Hostname = string()

Returns the local hostname. Never fails.

getifaddrs() ->
  {ok,
   [{Ifname :: string(),
     Ifoctxs :: getifaddrs_ifoctxs()}] |
    {error, posix()}

Returns a list of 2-tuples containing interface names and the interfaces’ addresses. Ifname is a Unicode string and Ifoctxs is a list of interface address description tuples. The interface address description tuples are documented under the type of the Ifoctxs value.

getifaddrs(Opts) -> {ok, [{Ifname, Ifoctxs}]} | {error, Posix}

Types:
  Opts = [{netns, Namespace}]
  Namespace = file:filename_all()
  Ifname = string()
  Ifoctxs = getifaddrs_ifoctxs()
  Posix = posix()

The same as getifaddrs/0 but the Option {netns, Namespace} sets a network namespace for the OS call, on platforms that supports that feature.

See the socket option {netns, Namespace} under setopts/2.

getopts(Socket, Options) -> {ok, OptionValues} | {error, posix()}

Types:
  Socket = socket()
  Options = [socket_getopt()]
  OptionValues = [socket_setopt() | gen_tcp:ptkoptions_value()]

Gets one or more options for a socket. For a list of available options, see setopts/2. See also the description for the type gen_tcp:ptkoptions_value().

The number of elements in the returned OptionValues list does not necessarily correspond to the number of options asked for. If the operating system fails to support an option, it is left out in the returned list. An error tuple is returned only when getting options for the socket is impossible (that is, the socket is closed or the buffer size in a raw request is too large). This behavior is kept for backward compatibility reasons.

A raw option request RawOptReq = {raw, Protocol, OptionNum, ValueSpec} can be used to get information about socket options not (explicitly) supported by the emulator. The use of raw socket options makes the
code non-portable, but allows the Erlang programmer to take advantage of unusual features present on a particular platform.

`RawOptReq` consists of tag `raw` followed by the protocol level, the option number, and either a binary or the size, in bytes, of the buffer in which the option value is to be stored. A binary is to be used when the underlying `getsockopt` requires `input` in the argument field. In this case, the binary size is to correspond to the required buffer size of the return value. The supplied values in a `RawOptReq` correspond to the second, third, and fourth/fifth parameters to the `getsockopt` call in the C socket API. The value stored in the buffer is returned as a binary `ValueBin`, where all values are coded in the native endianness.

Asking for and inspecting raw socket options require low-level information about the current operating system and TCP stack.

**Example:**

Consider a Linux machine where option `TCP_INFO` can be used to collect TCP statistics for a socket. Assume you are interested in field `tcpi_sacked` of `struct tcp_info` filled in when asking for `TCP_INFO`. To be able to access this information, you need to know the following:

- The numeric value of protocol level `IPPROTO_TCP`
- The numeric value of option `TCP_INFO`
- The size of `struct tcp_info`
- The size and offset of the specific field

By inspecting the headers or writing a small C program, it is found that `IPPROTO_TCP` is 6, `TCP_INFO` is 11, the structure size is 92 (bytes), the offset of `tcp_info_sacked` is 28 bytes, and the value is a 32-bit integer. The following code can be used to retrieve the value:

```erlang
get_tcpi_sacked(Sock) ->
    {ok,[[{raw,_,_,Info}]]} = inet:getopts(Sock,[{raw,6,11,92}]),
    <<_:28/binary,TcpiSacked:32/native,_/binary>> = Info,
    TcpiSacked.
```

Preferably, you would check the machine type, the operating system, and the Kernel version before executing anything similar to this code.

```erlang
getstat(Socket) -> {ok, OptionValues} | {error, posix()}
getstat(Socket, Options) -> {ok, OptionValues} | {error, posix()}
```

**Types:**

```erlang
Socket = socket()
Options = [stat_option()]
OptionValues = [{stat_option(), integer()}]
stat_option() =
    recv_cnt | recv_max | recv_avg | recv_oct | recv_dvi |
    send_cnt | send_max | send_avg | send_oct | send_pend
```

Gets one or more statistic options for a socket.

`getstat(Socket)` is equivalent to `getstat(Socket, [recv_avg, recv_cnt, recv_dvi, recv_max, recv_oct, send_avg, send_cnt, send_pend, send_max, send_oct])`.

The following options are available:

- `recv_avg`
  
  Average size of packets, in bytes, received by the socket.
recv_cnt
   Number of packets received by the socket.
recv_dvi
   Average packet size deviation, in bytes, received by the socket.
recv_max
   Size of the largest packet, in bytes, received by the socket.
recv_oct
   Number of bytes received by the socket.
send_avg
   Average size of packets, in bytes, sent from the socket.
send_cnt
   Number of packets sent from the socket.
send_pend
   Number of bytes waiting to be sent by the socket.
send_max
   Size of the largest packet, in bytes, sent from the socket.
send_oct
   Number of bytes sent from the socket.

i() -> ok
i(Proto :: socket_protocol()) -> ok
i(X1 :: socket_protocol(), Fs :: [atom()]) -> ok

Lists all TCP, UDP and SCTP sockets, including those that the Erlang runtime system uses as well as those created by the application.

The following options are available:

port
   The internal index of the port.
module
   The callback module of the socket.
recv
   Number of bytes received by the socket.
sent
   Number of bytes sent from the socket.
owner
   The socket owner process.
local_address
   The local address of the socket.
foreign_address
The address and port of the other end of the connection.

state
The connection state.

type
STREAM or DGRAM or SEQPACKET.

ntoa(IpAddress) -> Address | {error, einval}
Types:
Address = string()
IpAddress = ip_address()
 Parses an ip_address() and returns an IPv4 or IPv6 address string.

parse_address(Address) -> {ok, IPAddress} | {error, einval}
Types:
Address = string()
 IPAddress = ip_address()
 Parses an IPv4 or IPv6 address string and returns an ip4_address() or ip6_address(). Accepts a shortened IPv4 address string.

parse_ipv4_address(Address) -> {ok, IPv4Address} | {error, einval}
Types:
Address = string()
 IPv4Address = ip_address()
 Parses an IPv4 address string and returns an ip4_address(). Accepts a shortened IPv4 address string.

parse_ipv4strict_address(Address) -> {ok, IPv4Address} | {error, einval}
Types:
Address = string()
 IPv4Address = ip_address()
 Parses an IPv4 address string containing four fields, that is, not shortened, and returns an ip4_address().

parse_ipv6_address(Address) -> {ok, IPv6Address} | {error, einval}
Types:
Address = string()
 IPv6Address = ip_address()
 Parses an IPv6 address string and returns an ip6_address(). If an IPv4 address string is specified, an IPv4-mapped IPv6 address is returned.

parse_ipv6strict_address(Address) -> {ok, IPv6Address} | {error, einval}
Types:
Address = string()
IPv6Address = ip_address()

Parses an IPv6 address string and returns an ip6_address(). Does not accept IPv4 addresses.

ipv4_mapped_ipv6_address(X1 :: ip_address()) -> ip_address()

Convert an IPv4 address to an IPv4-mapped IPv6 address or the reverse. When converting from an IPv6 address all but the 2 low words are ignored so this function also works on some other types of addresses than IPv4-mapped.

parse_strict_address(Address) -> {ok, IPAddress} | {error, einval}

Types:
  Address = string()
  IPAddress = ip_address()

Parses an IPv4 or IPv6 address string and returns an ip4_address() or ip6_address(). Does not accept a shortened IPv4 address string.

peername(Socket :: socket()) ->
  {ok,
   [ip_address(), port_number()] |
   returned_non_ip_address()} |
  {error, posix()}  

Returns the address and port for the other end of a connection.

Notice that for SCTP sockets, this function returns only one of the peer addresses of the socket. Function peernames/1,2 returns all.

peernames(Socket :: socket()) ->
  {ok,
   [{ip_address(), port_number()} |
    returned_non_ip_address()]} |
  {error, posix()}

Equivalent to peernames(Socket, 0).

Notice that the behavior of this function for an SCTP one-to-many style socket is not defined by the SCTP Sockets API Extensions.

peernames(Socket, Assoc) ->
  {ok, [{Address, Port}]} | {error, posix()}

Types:
  Socket = socket()
  Assoc = #sctp_assoc_change{} | gen_sctp:assoc_id()
  Address = ip_address()
  Port = integer() >= 0

Returns a list of all address/port number pairs for the other end of an association Assoc of a socket.

This function can return multiple addresses for multihomed sockets, such as SCTP sockets. For other sockets it returns a one-element list.

Notice that parameter Assoc is by the SCTP Sockets API Extensions defined to be ignored for one-to-one style sockets. What the special value 0 means, hence its behavior for one-to-many style sockets, is unfortunately undefined.
port(Socket) -> {ok, Port} | {error, any()}

Types:
  Socket = socket()
  Port = port_number()

Returns the local port number for a socket.

setopts(Socket, Options) -> ok | {error, posix()}

Types:
  Socket = socket()
  Options = [socket_setopt()]

Sets one or more options for a socket.

The following options are available:

{active, true | false | once | N}

If the value is true, which is the default, everything received from the socket is sent as messages to the receiving process.

If the value is false (passive mode), the process must explicitly receive incoming data by calling gen_tcp:recv/2,3, gen_udp:recv/2,3, or gen_sctp:recv/1,2 (depending on the type of socket).

If the value is once {(active, once)}, one data message from the socket is sent to the process. To receive one more message, setopts/2 must be called again with option {active, once}.

If the value is an integer N in the range -32768 to 32767 (inclusive), the value is added to the socket's count of data messages sent to the controlling process. A socket's default message count is 0. If a negative value is specified, and its magnitude is equal to or greater than the socket's current message count, the socket's message count is set to 0. Once the socket's message count reaches 0, either because of sending received data messages to the process or by being explicitly set, the process is then notified by a special message, specific to the type of socket, that the socket has entered passive mode. Once the socket enters passive mode, to receive more messages setopts/2 must be called again to set the socket back into an active mode.

When using {active, once} or {active, N}, the socket changes behavior automatically when data is received. This can be confusing in combination with connection-oriented sockets (that is, gen_tcp), as a socket with {active, false} behavior reports closing differently than a socket with {active, true} behavior. To simplify programming, a socket where the peer closed, and this is detected while in {active, false} mode, still generates message tcp_closed,Socket when set to {active, once}, {active, true}, or {active, N} mode. It is therefore safe to assume that message tcp_closed,Socket, possibly followed by socket port termination (depending on option exit_on_close) eventually appears when a socket changes back and forth between {active, true} and {active, false} mode. However, when peer closing is detected it is all up to the underlying TCP/IP stack and protocol.

Notice that {active, true} mode provides no flow control; a fast sender can easily overflow the receiver with incoming messages. The same is true for {active, N} mode, while the message count is greater than zero.

Use active mode only if your high-level protocol provides its own flow control (for example, acknowledging received messages) or the amount of data exchanged is small. {active, false} mode, use of the {active, once} mode, or {active, N} mode with values of N appropriate for the application provides flow control. The other side cannot send faster than the receiver can read.

{broadcast, Boolean} (UDP sockets)

Enables/disables permission to send broadcasts.
{buffer, Size}

The size of the user-level buffer used by the driver. Not to be confused with options sndbuf and recbuf, which correspond to the Kernel socket buffers. For TCP it is recommended to have \( \text{val(buffer)} \geq \text{val(recbuf)} \) to avoid performance issues because of unnecessary copying. For UDP the same recommendation applies, but the max should not be larger than the MTU of the network path. \( \text{val(buffer)} \) is automatically set to the above maximum when recbuf is set. However, as the size set for recbuf usually become larger, you are encouraged to use getopt/2 to analyze the behavior of your operating system.

Note that this is also the maximum amount of data that can be received from a single recv call. If you are using higher than normal MTU consider setting buffer higher.

{delay_send, Boolean}

Normally, when an Erlang process sends to a socket, the driver tries to send the data immediately. If that fails, the driver uses any means available to queue up the message to be sent whenever the operating system says it can handle it. Setting \{delay_send, true\} makes all messages queue up. The messages sent to the network are then larger but fewer. The option affects the scheduling of send requests versus Erlang processes instead of changing any real property of the socket. The option is implementation-specific. Defaults to false.

{deliver, port | term}

When \{active, true\}, data is delivered on the form \text{port:}\{S, \{data, \[H1,..Hsz | Data\]\}\} or \text{term:}\{tcp, S, \[H1..Hsz | Data\]\}.

{dontroute, Boolean}

Enables/disables routing bypass for outgoing messages.

{exit_on_close, Boolean}

This option is set to true by default.

The only reason to set it to false is if you want to continue sending data to the socket after a close is detected, for example, if the peer uses \text{gen_tcp:shutdown/2} to shut down the write side.

{header, Size}

This option is only meaningful if option binary was specified when the socket was created. If option header is specified, the first Size number bytes of data received from the socket are elements of a list, and the remaining data is a binary specified as the tail of the same list. For example, if Size == 2, the data received matches [Byte1,Byte2|Binary].

{high_msgq_watermark, Size}

The socket message queue is set to a busy state when the amount of data on the message queue reaches this limit. Notice that this limit only concerns data that has not yet reached the ERTS internal socket implementation. Defaults to 8 kB.

Senders of data to the socket are suspended if either the socket message queue is busy or the socket itself is busy.

For more information, see options low_msgq_watermark, high_watermark, and low_watermark.

Notice that distribution sockets disable the use of high_msgq_watermark and low_msgq_watermark. Instead use the distribution buffer busy limit, which is a similar feature.

{high_watermark, Size} (TCP/IP sockets)

The socket is set to a busy state when the amount of data queued internally by the ERTS socket implementation reaches this limit. Defaults to 8 kB.

Senders of data to the socket are suspended if either the socket message queue is busy or the socket itself is busy.

For more information, see options low_watermark, high_msgq_watermark, and low_msgq_watermark.
{ipv6_v6only, Boolean}

Restricts the socket to use only IPv6, prohibiting any IPv4 connections. This is only applicable for IPv6 sockets (option inet6).

On most platforms this option must be set on the socket before associating it to an address. It is therefore only reasonable to specify it when creating the socket and not to use it when calling function (setopts/2) containing this description.

The behavior of a socket with this option set to true is the only portable one. The original idea when IPv6 was new of using IPv6 for all traffic is now not recommended by FreeBSD (you can use {ipv6_v6only, false} to override the recommended system default value), forbidden by OpenBSD (the supported GENERIC kernel), and impossible on Windows (which has separate IPv4 and IPv6 protocol stacks). Most Linux distros still have a system default value of false. This policy shift among operating systems to separate IPv6 from IPv4 traffic has evolved, as it gradually proved hard and complicated to get a dual stack implementation correct and secure.

On some platforms, the only allowed value for this option is true, for example, OpenBSD and Windows. Trying to set this option to false, when creating the socket, fails in this case.

Setting this option on platforms where it does not exist is ignored. Getting this option with getopts/2 returns no value, that is, the returned list does not contain an {ipv6_v6only, _} tuple. On Windows, the option does not exist, but it is emulated as a read-only option with value true.

Therefore, setting this option to true when creating a socket never fails, except possibly on a platform where you have customized the kernel to only allow false, which can be doable (but awkward) on, for example, OpenBSD.

If you read back the option value using getopts/2 and get no value, the option does not exist in the host operating system. The behavior of both an IPv6 and an IPv4 socket listening on the same port, and for an IPv6 socket getting IPv4 traffic is then no longer predictable.

{keepalive, Boolean}(TCP/IP sockets)

Enables/disables periodic transmission on a connected socket when no other data is exchanged. If the other end does not respond, the connection is considered broken and an error message is sent to the controlling process. Defaults to disabled.

{linger, {true|false, Seconds}}

Determines the time-out, in seconds, for flushing unsent data in the close/1 socket call.

The first component is if linger is enabled, the second component is the flushing time-out, in seconds. There are 3 alternatives:

{false, _}

close/1 or shutdown/2 returns immediately, not waiting for data to be flushed, with closing happening in the background.

{true, 0}

Aborts the connection when it is closed. Discards any data still remaining in the send buffers and sends RST to the peer.

This avoids TCP's TIME_WAIT state, but leaves open the possibility that another "incarnation" of this connection being created.

{true, Time} when Time > 0

close/1 or shutdown/2 will not return until all queued messages for the socket have been successfully sent or the linger timeout (Time) has been reached.
{low_msgq_watermark, Size}

If the socket message queue is in a busy state, the socket message queue is set in a not busy state when the amount of data queued in the message queue falls below this limit. Notice that this limit only concerns data that has not yet reached the ERTS internal socket implementation. Defaults to 4 kB.

Senders that are suspended because of either a busy message queue or a busy socket are resumed when the socket message queue and the socket are not busy.

For more information, see options high_msgq_watermark, high_watermark, and low_watermark.

Notice that distribution sockets disable the use of high_msgq_watermark and low_msgq_watermark. Instead they use the distribution buffer busy limit, which is a similar feature.

{low_watermark, Size} (TCP/IP sockets)

If the socket is in a busy state, the socket is set in a not busy state when the amount of data queued internally by the ERTS socket implementation falls below this limit. Defaults to 4 kB.

Senders that are suspended because of a busy message queue or a busy socket are resumed when the socket message queue and the socket are not busy.

For more information, see options high_watermark, high_msgq_watermark, and low_msgq_watermark.

{mode, Mode :: binary | list}

Received Packet is delivered as defined by Mode.

{netns, Namespace :: file:filename_all()}

Sets a network namespace for the socket. Parameter Namespace is a filename defining the namespace, for example, "/var/run/netns/example", typically created by command ip netns add example. This option must be used in a function call that creates a socket, that is, gen_tcp:connect/3,4, gen_tcp:listen/2, gen_udp:open/1,2 or gen_sctp:open/0,1,2, and also getifaddrs/1.

This option uses the Linux-specific syscall setns(), such as in Linux kernel 3.0 or later, and therefore only exists when the runtime system is compiled for such an operating system.

The virtual machine also needs elevated privileges, either running as superuser or (for Linux) having capability CAP_SYS_ADMIN according to the documentation for setns(2). However, during testing also CAP_SYS_PTRACE and CAP_DAC_READ_SEARCH have proven to be necessary.

Example:

```
setcap cap_sys_admin,cap_sys_ptrace,cap_dac_read_search+epi beam.smp
```

Notice that the filesystem containing the virtual machine executable (beam.smp in the example) must be local, mounted without flag nosetuid, support extended attributes, and the kernel must support file capabilities. All this runs out of the box on at least Ubuntu 12.04 LTS, except that SCTP sockets appear to not support network namespaces.

Namespace is a filename and is encoded and decoded as discussed in module file, with the following exceptions:

- Emulator flag +fnu is ignored.
- getopt/2 for this option returns a binary for the filename if the stored filename cannot be decoded. This is only to occur if you set the option using a binary that cannot be decoded with the emulator's filename encoding: file:native_name_encoding/0.

{bind_to_device, Ifname :: binary()}

Binds a socket to a specific network interface. This option must be used in a function call that creates a socket, that is, gen_tcp:connect/3,4, gen_tcp:listen/2, gen_udp:open/1,2, or gen_sctp:open/0,1,2.
Unlike `getifaddrs/0`, `Ifname` is encoded a binary. In the unlikely case that a system is using non-7-bit-ASCII characters in network device names, special care has to be taken when encoding this argument.

This option uses the Linux-specific socket option `SO_BINDTODEVICE`, such as in Linux kernel 2.0.30 or later, and therefore only exists when the runtime system is compiled for such an operating system.

Before Linux 3.8, this socket option could be set, but could not retrieved with `getopts/2`. Since Linux 3.8, it is readable.

The virtual machine also needs elevated privileges, either running as superuser or (for Linux) having capability `CAP_NET_RAW`.

The primary use case for this option is to bind sockets into Linux VRF instances.

```plaintext
list
  Received Packet is delivered as a list.

binary
  Received Packet is delivered as a binary.

{nodelay, Boolean}(TCP/IP sockets)
  If Boolean == true, option TCP_NODELAY is turned on for the socket, which means that also small amounts of data are sent immediately.

{nopush, Boolean}(TCP/IP sockets)
  This translates to TCP_NOPUSH on BSD and to TCP_CORK on Linux.
  If Boolean == true, the corresponding option is turned on for the socket, which means that small amounts of data are accumulated until a full MSS-worth of data is available or this option is turned off.

Note that while TCP_NOPUSH socket option is available on OSX, its semantics is very different (e.g., unsetting it does not cause immediate send of accumulated data). Hence, nopush option is intentionally ignored on OSX.

{packet, PacketType}(TCP/IP sockets)
  Defines the type of packets to use for a socket. Possible values:
  raw | 0
    No packaging is done.
  1 | 2 | 4
    Packets consist of a header specifying the number of bytes in the packet, followed by that number of bytes. The header length can be one, two, or four bytes, and containing an unsigned integer in big-endian byte order. Each send operation generates the header, and the header is stripped off on each receive operation.
    The 4-byte header is limited to 2Gb.
  asn1 | cdr | sunrm | fcgi | tpkt | line
    These packet types only have effect on receiving. When sending a packet, it is the responsibility of the application to supply a correct header. On receiving, however, one message is sent to the controlling process for each complete packet received, and, similarly, each call to `gen_tcp:recv/2,3` returns one complete packet. The header is not stripped off.
    The meanings of the packet types are as follows:
    • asn1 - ASN.1 BER
    • sunrm - Sun's RPC encoding
    • cdr - CORBA (GIOP 1.1)
    • fcgi - Fast CGI
```
• tpkt - TPKT format [RFC1006]
• line - Line mode, a packet is a line-terminated with newline, lines longer than the receive buffer are truncated

http | http_bin

The Hypertext Transfer Protocol. The packets are returned with the format according to HttpPacket described in erlang:decode_packet/3 in ERTS. A socket in passive mode returns {ok, HttpPacket} from gen_tcp:recv while an active socket sends messages like {http, Socket, HttpPacket}.

httpth | httpth_bin

These two types are often not needed, as the socket automatically switches from http/http_bin to http/httpth_bin internally after the first line is read. However, there can be occasions when they are useful, such as parsing trailers from chunked encoding.

{packet_size, Integer}(TCP/IP sockets)

Sets the maximum allowed length of the packet body. If the packet header indicates that the length of the packet is longer than the maximum allowed length, the packet is considered invalid. The same occurs if the packet header is too large for the socket receive buffer.

For line-oriented protocols (line, http*), option packet_size also guarantees that lines up to the indicated length are accepted and not considered invalid because of internal buffer limitations.

{line_delimiter, Char}(TCP/IP sockets)

Sets the line delimiting character for line-oriented protocols (line). Defaults to \n.

{raw, Protocol, OptionNum, ValueBin}

See below.

{read_packets, Integer}(UDP sockets)

Sets the maximum number of UDP packets to read without intervention from the socket when data is available. When this many packets have been read and delivered to the destination process, new packets are not read until a new notification of available data has arrived. Defaults to 5. If this parameter is set too high, the system can become unresponsive because of UDP packet flooding.

{recbuf, Size}

The minimum size of the receive buffer to use for the socket. You are encouraged to use getopt/2 to retrieve the size set by your operating system.

{recvtclass, Boolean}

If set to true activates returning the received TCLASS value on platforms that implements the protocol IPPROTO_IPV6 option IPV6_RECVCLASS or IPV6_2292_RECVCLASS for the socket. The value is returned as a {tclass, TCLASS} tuple regardless of if the platform returns an IPV6_TCLASS or an IPV6_RECVCLASS CMSG value.

For packet oriented sockets that supports receiving ancillary data with the payload data (gen_udp and gen_sctp), the TCLASS value is returned in an extended return tuple contained in an ancillary data list. For stream oriented sockets (gen_tcp) the only way to get the TCLASS value is if the platform supports the pktoptions option.

{recvtos, Boolean}

If set to true activates returning the received TOS value on platforms that implements the protocol IPPROTO_IP option IP_RECVTOS for the socket. The value is returned as a {tos, TOS} tuple regardless of if the platform returns an IP_TOS or an IP_RECVTOS CMSG value.
For packet oriented sockets that supports receiving ancillary data with the payload data (gen_udp and gen_sctp), the TOS value is returned in an extended return tuple contained in an ancillary data list. For stream oriented sockets (gen_tcp) the only way to get the TOS value is if the platform supports the pktoptions option.

{recvttl, Boolean}
If set to true activates returning the received TTL value on platforms that implements the protocol IPPROTO_IP option IP_RECVTTL for the socket. The value is returned as a {ttl, TTL} tuple regardless of if the platform returns an IP_TTL or an IP_RECVTTL CMSG value.

For packet oriented sockets that supports receiving ancillary data with the payload data (gen_udp and gen_sctp), the TTL value is returned in an extended return tuple contained in an ancillary data list. For stream oriented sockets (gen_tcp) the only way to get the TTL value is if the platform supports the pktoptions option.

{reuseaddr, Boolean}
Allows or disallows local reuse of port numbers. By default, reuse is disallowed.

{send_timeout, Integer}
Only allowed for connection-oriented sockets.
Specifies a longest time to wait for a send operation to be accepted by the underlying TCP stack. When the limit is exceeded, the send operation returns {error,timeout}. How much of a packet that got sent is unknown; the socket is therefore to be closed whenever a time-out has occurred (see send_timeout_close below). Defaults to infinity.

{send_timeout_close, Boolean}
Only allowed for connection-oriented sockets.
Used together with send_timeout to specify whether the socket is to be automatically closed when the send operation returns {error,timeout}. The recommended setting is true, which automatically closes the socket. Defaults to false because of backward compatibility.

{show_econnreset, Boolean}(TCP/IP sockets)
When this option is set to false, which is default, an RST received from the TCP peer is treated as a normal close (as though an FIN was sent). A caller to gen_tcp:recv/2 gets {error, closed}. In active mode, the controlling process receives a {tcp_closed, Socket} message, indicating that the peer has closed the connection.
Setting this option to true allows you to distinguish between a connection that was closed normally, and one that was aborted (intentionally or unintentionally) by the TCP peer. A call to gen_tcp:recv/2 returns {error, econnreset}. In active mode, the controlling process receives a {tcp_error, Socket, econnreset} message before the usual {tcp_closed, Socket}, as is the case for any other socket error. Calls to gen_tcp:send/2 also returns {error, econnreset} when it is detected that a TCP peer has sent an RST.
A connected socket returned from gen_tcp:accept/1 inherits the show_econnreset setting from the listening socket.

{sndbuf, Size}
The minimum size of the send buffer to use for the socket. You are encouraged to use getopt/2, to retrieve the size set by your operating system.
{priority, Integer}

Sets the SO_PRIORITY socket level option on platforms where this is implemented. The behavior and allowed range varies between different systems. The option is ignored on platforms where it is not implemented. Use with caution.

{tos, Integer}

Sets IP_TOS IP level options on platforms where this is implemented. The behavior and allowed range varies between different systems. The option is ignored on platforms where it is not implemented. Use with caution.

{tclass, Integer}

Sets IPV6_TCLASS IP level options on platforms where this is implemented. The behavior and allowed range varies between different systems. The option is ignored on platforms where it is not implemented. Use with caution.

In addition to these options, raw option specifications can be used. The raw options are specified as a tuple of arity four, beginning with tag raw, followed by the protocol level, the option number, and the option value specified as a binary. This corresponds to the second, third, and fourth arguments to the setsockopt call in the C socket API. The option value must be coded in the native endianess of the platform and, if a structure is required, must follow the structure alignment conventions on the specific platform.

Using raw socket options requires detailed knowledge about the current operating system and TCP stack.

Example:

This example concerns the use of raw options. Consider a Linux system where you want to set option TCP_LINGER2 on protocol level IPPROTO_TCP in the stack. You know that on this particular system it defaults to 60 (seconds), but you want to lower it to 30 for a particular socket. Option TCP_LINGER2 is not explicitly supported by inet, but you know that the protocol level translates to number 6, the option number to number 8, and the value is to be specified as a 32-bit integer. You can use this code line to set the option for the socket named Sock:

```
inet:setopts(Sock, [{raw, 6, 8, <<30:32/native>>}]),
```

As many options are silently discarded by the stack if they are specified out of range; it can be a good idea to check that a raw option is accepted. The following code places the value in variable TcpLinger2:

```
{ok, [{raw, 6, 8, <<TcpLinger2:32/native>>}]} = inet:getopts(Sock, [{raw, 6, 8, 4}]),
```

Code such as these examples is inherently non-portable, even different versions of the same OS on the same platform can respond differently to this kind of option manipulation. Use with care.

Notice that the default options for TCP/IP sockets can be changed with the Kernel configuration parameters mentioned in the beginning of this manual page.

sockname(Socket :: socket()) ->
    {ok, {ip_address(), port_number()}} |
    returned_non_ip_address() |
    {error, posix()}

Returns the local address and port number for a socket.

Notice that for SCTP sockets this function returns only one of the socket addresses. Function socknames/1,2 returns all.

socknames(Socket :: socket()) ->
    {ok, [{ip_address(), port_number()}] |
socknames(Socket, Assoc) ->
    {ok, [{Address, Port}]} | {error, posix()}

Types:
    Socket = socket()
    Assoc = #sctp_assoc_change{} | gen_sctp:assoc_id()
    Address = ip_address()
    Port = integer() >= 0

Returns a list of all local address/port number pairs for a socket for the specified association Assoc.

This function can return multiple addresses for multihomed sockets, such as SCTP sockets. For other sockets it returns a one-element list.

Notice that parameter Assoc is by the SCTP Sockets API Extensions defined to be ignored for one-to-one style sockets. For one-to-many style sockets, the special value 0 is defined to mean that the returned addresses must be without any particular association. How different SCTP implementations interpret this varies somewhat.

POSIX Error Codes

- e2big - Too long argument list
- eacces - Permission denied
- eaddrinuse - Address already in use
- eaddrnotavail - Cannot assign requested address
- eadv - Advertise error
- eafnosupport - Address family not supported by protocol family
- eagain - Resource temporarily unavailable
- ealign - EALIGN
- ealready - Operation already in progress
- ebade - Bad exchange descriptor
- ebadf - Bad file number
- ebadfd - File descriptor in bad state
- ebadmsg - Not a data message
- ebadr - Bad request descriptor
- ebadrpc - Bad RPC structure
- ebadrqc - Bad request code
- ebadslt - Invalid slot
- ebfont - Bad font file format
- ebfru - File busy
- echild - No children
- echrng - Channel number out of range
- ecomm - Communication error on send
- econnaborted - Software caused connection abort
- econnrefused - Connection refused
• econnreset - Connection reset by peer
• edeadlk - Resource deadlock avoided
• edeadlock - Resource deadlock avoided
• edestaddrreq - Destination address required
• edirty - Mounting a dirty fs without force
• edom - Math argument out of range
• edotdot - Cross mount point
• edquot - Disk quota exceeded
• eduppkg - Duplicate package name
• eexist - File already exists
• efault - Bad address in system call argument
• efbig - File too large
• ehostdown - Host is down
• ehostunreach - Host is unreachable
• eidrm - Identifier removed
• einit - Initialization error
• einprogress - Operation now in progress
• eintr - Interrupted system call
• einval - Invalid argument
• eio - I/O error
• eisconn - Socket is already connected
• eisdirt - Illegal operation on a directory
• eisnam - Is a named file
• el2hlt - Level 2 halted
• el2nsync - Level 2 not synchronized
• el3hlt - Level 3 halted
• el3rst - Level 3 reset
• elbin - ELBIN
• elibacc - Cannot access a needed shared library
• elibbad - Accessing a corrupted shared library
• elibexec - Cannot exec a shared library directly
• elibmax - Attempting to link in more shared libraries than system limit
• elibscn - .lib section in a.out corrupted
• eln rng - Link number out of range
• e loop - Too many levels of symbolic links
• emfile - Too many open files
• emlink - Too many links
• emsgsize - Message too long
• emultihop - Multihop attempted
• enametoolong - Filename too long
• enavail - Unavailable
• enet - ENET
• enetdowm - Network is down
- enetreset - Network dropped connection on reset
- enetunreach - Network is unreachable
- enfile - File table overflow
- enoano - Anode table overflow
- enobufs - No buffer space available
- enocsi - No CSI structure available
- enodata - No data available
- enodev - No such device
- enoent - No such file or directory
- enoexec - Exec format error
- enolck - No locks available
- enolink - Link has been severed
- enomem - Not enough memory
- enomsg - No message of desired type
- enonet - Machine is not on the network
- enopkg - Package not installed
- enoprotoopt - Bad protocol option
- enospc - No space left on device
- enosr - Out of stream resources or not a stream device
- enosys - Function not implemented
- enotblk - Block device required
- enotconn - Socket is not connected
- enotdir - Not a directory
- enotempty - Directory not empty
- enotnam - Not a named file
- enotsock - Socket operation on non-socket
- enotsup - Operation not supported
- enotty - Inappropriate device for ioctl
- enotuniq - Name not unique on network
- enxio - No such device or address
- eopnotsupp - Operation not supported on socket
- eperm - Not owner
- epfnosupport - Protocol family not supported
- epipe - Broken pipe
- eproclim - Too many processes
- eprocunavail - Bad procedure for program
- eprogmismatch - Wrong program version
- eprogunavail - RPC program unavailable
- eproto - Protocol error
- eprotonosupport - Protocol not supported
- epotype - Wrong protocol type for socket
- erange - Math result unrepresentable
• erefused - EREFUSED
• eremchg - Remote address changed
• eremdev - Remote device
• eremote - Pathname hit remote filesystem
• eremoteio - Remote I/O error
• eremoterelease - EREMOTERELEASE
• erofs - Read-only filesystem
• erpcmismatch - Wrong RPC version
• erremote - Object is remote
• eshutdown - Cannot send after socket shutdown
• esocktnosupport - Socket type not supported
• espipe - Invalid seek
• esrch - No such process
• esrmnt - Srmount error
• estale - Stale remote file handle
• esuccess - Error 0
• etime - Timer expired
• etimedout - Connection timed out
• etoomanyrefs - Too many references
• etxtbsy - Text file or pseudo-device busy
• euclean - Structure needs cleaning
• eunatch - Protocol driver not attached
• eusers - Too many users
• eversion - Version mismatch
• ewouldblock - Operation would block
• exdev - Cross-domain link
• exfull - Message tables full
• nxdomain - Hostname or domain name cannot be found
This module performs DNS name resolving to recursive name servers.
See also ERTS User's Guide: Inet Configuration for more information about how to configure an Erlang runtime system for IP communication, and how to enable this DNS client by defining 'dns' as a lookup method. The DNS client then acts as a backend for the resolving functions in inet.
This DNS client can resolve DNS records even if it is not used for normal name resolving in the node.
This is not a full-fledged resolver, only a DNS client that relies on asking trusted recursive name servers.

Name Resolving
UDP queries are used unless resolver option usevc is true, which forces TCP queries. If the query is too large for UDP, TCP is used instead. For regular DNS queries, 512 bytes is the size limit.
When EDNS is enabled (resolver option edns is set to the EDNS version (that is, 0 instead of false), resolver option udp_payload_size sets the limit. If a name server replies with the TC bit set (truncation), indicating that the answer is incomplete, the query is retried to that name server using TCP. Resolver option udp_payload_size also sets the advertised size for the maximum allowed reply size, if EDNS is enabled, otherwise the name server uses the limit 512 bytes. If the reply is larger, it gets truncated, forcing a TCP requery.
For UDP queries, resolver options timeout and retry control retransmission. Each name server in the nameservers list is tried with a time-out of timeout/retry. Then all name servers are tried again, doubling the time-out, for a total of retry times.
For queries not using the search list, if the query to all nameservers results in {error, nxdomain} or an empty answer, the same query is tried for alt_nameservers.

Resolver Types
The following data types concern the resolver:

Data Types
res_option() =
  {alt_nameservers, [nameserver()]} |
  {edns, 0 | false} |
  {inet6, boolean()} |
  {nameservers, [nameserver()]} |
  {recurse, boolean()} |
  {retry, integer()} |
  {timeout, integer()} |
  {udp_payload_size, integer()} |
  {usevc, boolean()}
nameserver() = {inet:ip_address(), Port :: 1..65535}
res_error() =
  formerr | qfmterror | servfail | nxdomain | notimp | refused |
  badvers | timeout

DNS Types
The following data types concern the DNS client:
Data Types

dns_name() = string()

A string with no adjacent dots.

rr_type() =
    a | aaaa | cname | gid | hinfo | ns | mb | md | mg | mf |
    minfo | mx | naptr | null | ptr | soa | spf | srv | txt |
    uid | uinfo | unspec | wks

dns_class() = in | chaos | hs | any

dns_msg() = term()

This is the start of a hierarchy of opaque data structures that can be examined with access functions in inet_dns, which return lists of {Field, Value} tuples. The arity 2 functions only return the value for a specified field.
dns_msg() = DnsMsg
   inet_dns:msg(DnsMsg) ->
      [ {header, dns_header()}
      | {qdlist, dns_query()}
      | {anlist, dns_rr()}
      | {nslist, dns_rr()}
      | {arlist, dns_rr()} ]
   inet_dns:msg(DnsMsg, header) -> dns_header() % for example
   inet_dns:msg(DnsMsg, Field) -> Value

dns_header() = DnsHeader
   inet_dns:header(DnsHeader) ->
      [ {id, integer()}
      | {qr, boolean()}
      | {opcode, query | iquery | status | integer()}
      | {aa, boolean()}
      | {tc, boolean()}
      | {rd, boolean()}
      | {ra, boolean()}
      | {pr, boolean()}
      | {rcode, integer(0..16)} ]
   inet_dns:header(DnsHeader, Field) -> Value

dns_query() = DnsQuery
   inet_dns:dns_query(DnsQuery) ->
      [ {domain, dns_name()}
      | {type, query_type()}
      | {class, dns_class()} ]
   inet_dns:dns_query(DnsQuery, Field) -> Value

dns_rr() = DnsRr
   inet_dns:rr(DnsRr) -> DnsRrFields | DnsRrOptFields
   DnsRrFields = [ {domain, dns_name()}
      | {type, rr_type()}
      | {class, dns_class()}
      | {ttl, integer()}
      | {data, dns_data()} ]
   DnsRrOptFields = [ {domain, dns_name()}
      | {type, opt}
      | {udp_payload_size, integer()}
      | {ext_rcode, integer()}
      | {version, integer()}
      | {z, integer()}
      | {data, dns_data()} ]
   inet_dns:rr(DnsRr, Field) -> Value

There is an information function for the types above:

   inet_dns:record_type(dns_msg()) -> msg;
   inet_dns:record_type(dns_header()) -> header;
   inet_dns:record_type(dns_query()) -> dns_query;
   inet_dns:record_type(dns_rr()) -> rr;
   inet_dns:record_type(_) -> undefined.

So, inet_dns:(inet_dns:record_type(X))(X) converts any of these data structures into a (Field,Value) list.

dns_data() =
   dns_name() |
   inet:ip4_address() |

inet:ip6_address() | {MName :: dns_name(),
    RName :: dns_name(),
    Serial :: integer(),
    Refresh :: integer(),
    Retry :: integer(),
    Expiry :: integer(),
    Minimum :: integer()} |
{inet:ip4_address(), Proto :: integer(), BitMap :: binary()} |
{CpuString :: string(), OsString :: string()} |
{RM :: dns_name(), EM :: dns_name()} |
{Prio :: integer(), dns_name()} |
{Prio :: integer(),
    Weight :: integer(),
    Port :: integer(),
    dns_name()} |
{Order :: integer(),
    Preference :: integer(),
    Flags :: string(),
    Services :: string(),
    RegExp :: string(),
    dns_name()} |
[string()] |
binary()

RegExp is a string with characters encoded in the UTF-8 coding standard.

Exports

getbyname(Name, Type) -> {ok, Hostent} | {error, Reason}
getbyname(Name, Type, Timeout) -> {ok, Hostent} | {error, Reason}

Types:
Name = dns_name()
Type = rr_type()
Timeout = timeout()
Hostent = inet:hostent()
Reason = inet:posix() | res_error()

Resolves a DNS record of the specified type for the specified host, of class in. Returns, on success, a hostent() record with dns_data() elements in the address list field.

This function uses resolver option search that is a list of domain names. If the name to resolve contains no dots, it is prepended to each domain name in the search list, and they are tried in order. If the name contains dots, it is first tried as an absolute name and if that fails, the search list is used. If the name has a trailing dot, it is supposed to be an absolute name and the search list is not used.

gethostbyaddr(Address) -> {ok, Hostent} | {error, Reason}
gethostbyaddr(Address, Timeout) -> {ok, Hostent} | {error, Reason}

Types:
Address = inet:ip_address()
Timeout = timeout()
Hostent = inet:hostent()
Reason = inet:posix() | res_error()

Backend functions used by inet: gethostbyaddr/1.

gethostbyname(Name) -> {ok, Hostent} | {error, Reason}
gethostbyname(Name, Family) -> {ok, Hostent} | {error, Reason}
gethostbyname(Name, Family, Timeout) ->
    {ok, Hostent} | {error, Reason}

Types:
    Name = dns_name()
    Hostent = inet:hostent()
    Timeout = timeout()
    Family = inet:address_family()
    Reason = inet:posix() | res_error()

Backend functions used by inet: gethostbyname/1,2.

This function uses resolver option search just like getbyname/2,3.
If resolver option inet6 is true, an IPv6 address is looked up.

lookup(Name, Class, Type) -> [dns_data()]
lookup(Name, Class, Type, Opts) -> [dns_data()]
lookup(Name, Class, Type, Opts, Timeout) -> [dns_data()]

Types:
    Name = dns_name() | inet:ip_address()
    Class = dns_class()
    Type = rr_type()
    Opts = [res_option() | verbose]
    Timeout = timeout()

Resolves the DNS data for the record of the specified type and class for the specified name. On success, filters out the answer records with the correct Class and Type, and returns a list of their data fields. So, a lookup for type any gives an empty answer, as the answer records have specific types that are not any. An empty answer or a failed lookup returns an empty list.

Calls resolve/* with the same arguments and filters the result, so Opts is described for those functions.

resolve(Name, Class, Type) -> {ok, dns_msg()} | Error
resolve(Name, Class, Type, Opts) -> {ok, dns_msg()} | Error
resolve(Name, Class, Type, Opts, Timeout) ->
    {ok, dns_msg()} | Error

Types:
Resolves a DNS record of the specified type and class for the specified name. The returned \texttt{dns\_msg()} can be examined using access functions in \texttt{inet\_db}, as described in section in DNS Types.

If \texttt{Name} is an \texttt{ip\_address()}, the domain name to query for is generated as the standard reverse ".IN-ADDR.ARPA." name for an IPv4 address, or the ".IP6.ARPA." name for an IPv6 address. In this case, you most probably want to use \texttt{Class = in} and \texttt{Type = ptr}, but it is not done automatically.

\texttt{Opts} overrides the corresponding resolver options. If option \texttt{nameservers} is specified, it is assumed that it is the complete list of name serves, so resolver option \texttt{alt\_nameserves} is ignored. However, if option \texttt{alt\_nameserves} is also specified to this function, it is used.

Option \texttt{verbose} (or rather \texttt{verbose,true}) causes diagnostics printout through \texttt{io:format/2} of queries, replies retransmissions, and so on, similar to from utilities, such as \texttt{dig} and \texttt{nslookup}.

If \texttt{Opt} is any atom, it is interpreted as \texttt{(Opt,true)} unless the atom string starts with \texttt{"no"}, making the interpretation \texttt{(Opt,false)}. For example, \texttt{usevc} is an alias for \texttt{(usevc,true)} and \texttt{nousevc} is an alias for \texttt{(usevc,false)}.

Option \texttt{inet6} has no effect on this function. You probably want to use \texttt{Type = a | aaaa} instead.

**Example**

This access functions example shows how \texttt{lookup/3} can be implemented using \texttt{resolve/3} from outside the module:

```erlang
example_lookup(Name, Class, Type) ->
case inet_res:resolve(Name, Class, Type) of
  {ok,Msg} ->
    inet_dns:rr(RR, data)
    ||
    RR <- inet_dns:msg(Msg, anlist),
    inet_dns:rr(RR, type) == Type,
    inet_dns:rr(RR, class) == Class;
  {error,_,_} ->
    []
end.
```

These are deprecated because the annoying double meaning of the name servers/time-out argument, and because they have no decent place for a resolver options list.

**Exports**

\texttt{nslookup(Name, Class, Type)} \to \{ok, \texttt{dns\_msg()}\} | \{error, Reason\}

\texttt{nslookup(Name, Class, Type, Timeout)} \to

\{ok, \texttt{dns\_msg()}\} | \{error, Reason\}

\texttt{nslookup(Name, Class, Type, Nameservers)} \to

\{ok, \texttt{dns\_msg()}\} | \{error, Reason\}

**Types:**
name = dns_name() | inet:ip_address()
class = dns_class()
type = rr_type()
timeout = timeout()
nameservers = [nameserver()]
reason = inet:posix() | res_error()

Resolves a DNS record of the specified type and class for the specified name.

nnslookup(Name, Class, Type, Nameservers) ->
    {ok, dns_msg()} | {error, Reason}
nnslookup(Name, Class, Type, Nameservers, Timeout) ->
    {ok, dns_msg()} | {error, Reason}

Types:
    name = dns_name() | inet:ip_address()
class = dns_class()
type = rr_type()
timeout = timeout()
nameservers = [nameserver()]
reason = inet:posix()

Resolves a DNS record of the specified type and class for the specified name.
**init**

Erlang module

This module is moved to the ERTS application.
logger

Erlang module

This module implements the main API for logging in Erlang/OTP. To create a log event, use the API functions or the log macros, for example:

```
?LOG_ERROR("error happened because: ~p", [Reason]).  % With macro
logger:error("error happened because: ~p", [Reason]).  % Without macro
```

To configure the Logger backend, use Kernel configuration parameters or configuration functions in the Logger API. By default, the Kernel application installs one log handler at system start. This handler is named `default`. It receives and processes standard log events produced by the Erlang runtime system, standard behaviours and different Erlang/OTP applications. The log events are by default printed to the terminal.

If you want your systems logs to be printed to a file instead, you must configure the default handler to do so. The simplest way is to include the following in your `sys.config`:

```
[{kernel,
  [{logger,
    [{handler, default, logger_std_h,
      #{config => #{file => "path/to/file.log"}}}]}
  ]}
```

For more information about:
- the Logger facility in general, see the User's Guide.
- how to configure Logger, see the Configuration section in the User's Guide.
- the built-in handlers, see `logger_std_h` and `logger_disk_log_h`.
- the built-in formatter, see `logger_formatter`.
- built-in filters, see `logger_filters`.

Note:

Since Logger is new in Erlang/OTP 21.0, we do reserve the right to introduce changes to the Logger API and functionality in patches following this release. These changes might or might not be backwards compatible with the initial version.

Data Types

```erl
filter() =
  {fun((log_event(), filter_arg()) -> filter_return()),
   filter_arg()}
```

A filter which can be installed as a handler filter, or as a primary filter in Logger.

```erl
filter_arg() = term()
```

The second argument to the filter fun.

```erl
filter_id() = atom()
```

A unique identifier for a filter.

```erl
filter_return() = stop | ignore | log_event()
```

The return value from the filter fun.
formatter_config() = #{atom() => term()}
Configuration data for the formatter. See logger_formatter(3) for an example of a formatter implementation.

handler_config() =
    #{id => handler_id(),
     config => term(),
     level => level() | all | none,
     module => module(),
     filter_default => log | stop,
     filters => [{filter_id(), filter()}],
     formatter => {module(), formatter_config()}}

Handler configuration data for Logger. The following default values apply:

• level => all
• filter_default => log
• filters => []
• formatter => {logger_formatter, DefaultFormatterConfig}

In addition to these, the following fields are automatically inserted by Logger, values taken from the two first parameters to add_handler/3:

• id => HandlerId
• module => Module

These are read-only and cannot be changed in runtime.

Handler specific configuration data is inserted by the handler callback itself, in a sub structure associated with the field named config. See the logger_std_h(3) and logger_disk_log_h(3) manual pages for information about the specific configuration for these handlers.

See the logger_formatter(3) manual page for information about the default configuration for this formatter.

handler_id() = atom()
A unique identifier for a handler instance.

level() =
    emergency | alert | critical | error | warning | notice | info | debug

The severity level for the message to be logged.

log_event() =
    #{level := level(),
     msg :=
     {io:format(), [term()]} |
     {report, report()} |
     {string, unicode:chardata()},
     meta := metadata()}

metadata() =
    #{pid => pid(),
     gl => pid(),
     time => timestamp(),
     mfa => {module(), atom(), integer() >= 0},
     file => file:filename(),
     line => integer() >= 0,
domain => [atom()],
report_cb => report_cb(),
atom() => term()}

Metadata for the log event.

Logger adds the following metadata to each log event:

- pid => self()
- gl => group_leader()
- time => logger:timestamp()

When a log macro is used, Logger also inserts location information:

- mfa => {?MODULE, ?FUNCTION_NAME, ?FUNCTION_ARITY}
- file => ?FILE
- line => ?LINE

You can add custom metadata, either by specifying a map as the last parameter to any of the log macros or the API functions, or by setting process metadata with set_process_metadata/1 or update_process_metadata/1.

Logger merges all the metadata maps before forwarding the log event to the handlers. If the same keys occur, values from the log call overwrite process metadata, which in turn overwrite values set by Logger.

The following custom metadata keys have special meaning:

- **domain**
  The value associated with this key is used by filters for grouping log events originating from, for example, specific functional areas. See logger_filters:domain/2 for a description of how this field can be used.

- **report_cb**
  If the log message is specified as a `report()`, the `report_cb` key can be associated with a fun (report callback) that converts the report to a format string and arguments, or directly to a string. See the type definition of `report_cb()`, and section Log Message in the User’s Guide for more information about report callbacks.

```erlang
msg_fun() =
    fun((term()) ->
        {io:format(), [term()]} |
        report() |
        unicode:chardata())
```

```erlang
olp_config() =
    #{
        sync_mode_qlen => integer() >= 0,
        drop_mode_qlen => integer() >= 1,
        flush_qlen => integer() >= 1,
        burst_limit_enable => boolean(),
        burst_limit_max_count => integer() >= 1,
        burst_limit_window_time => integer() >= 1,
        overload_kill_enable => boolean(),
        overload_kill_qlen => integer() >= 1,
        overload_kill_mem_size => integer() >= 1,
        overload_kill_restart_after => integer() >= 0 | infinity}
```

```erlang
primary_config() =
    #{
        level => level() | all | none,
        filter_default => log | stop,
```
Primary configuration data for Logger. The following default values apply:

- level => info
- filter_default => log
- filters => []

report() = map() | {[atom(), term()]}  
report_cb() = 
  fun((report()) -> {io:format(), [term()]} | fun((report(), report_cb_config()) -> unicode:chardata())

A fun which converts a report() to a format string and arguments, or directly to a string. See section Log Message in the User's Guide for more information.

report_cb_config() =
  #{depth := integer() >= 1 | unlimited,
  chars_limit := integer() >= 1 | unlimited,
  single_line := boolean()}

timestamp() = integer()

A timestamp produced with logger:timestamp().

Macros

The following macros are defined in logger.hrl, which is included in a module with the directive

```
-include_lib("kernel/include/logger.hrl").
```

- ?LOG_EMERGENCY(StringOrReport[, Metadata])
- ?LOG_EMERGENCY(FunOrFormat, Args[, Metadata])
- ?LOG_ALERT(StringOrReport[, Metadata])
- ?LOG_ALERT(FunOrFormat, Args[, Metadata])
- ?LOG_CRITICAL(StringOrReport[, Metadata])
- ?LOG_CRITICAL(FunOrFormat, Args[, Metadata])
- ?LOG_ERROR(StringOrReport[, Metadata])
- ?LOG_ERROR(FunOrFormat, Args[, Metadata])
- ?LOG_WARNING(StringOrReport[, Metadata])
- ?LOG_WARNING(FunOrFormat, Args[, Metadata])
- ?LOG_NOTICE(StringOrReport[, Metadata])
- ?LOG_NOTICE(FunOrFormat, Args[, Metadata])
- ?LOG_INFO(StringOrReport[, Metadata])
- ?LOG_INFO(FunOrFormat, Args[, Metadata])
- ?LOG_DEBUG(StringOrReport[, Metadata])
- ?LOG_DEBUG(FunOrFormat, Args[, Metadata])
- ?LOG(Level, StringOrReport[, Metadata])
- ?LOG(Level, FunOrFormat, Args[, Metadata])

All macros expand to a call to Logger, where Level is taken from the macro name, or from the first argument in the case of the ?LOG macro. Location data is added to the metadata as described under the metadata() type definition.
The call is wrapped in a case statement and will be evaluated only if Level is equal to or below the configured log level.

**Exports**

```plaintext
emergency(StringOrReport[,Metadata])
emergency(Format,Args[,Metadata])
emergency(Fun,FunArgs[,Metadata])
Equivalent to log(emergency,...).
```

```plaintext
alert(StringOrReport[,Metadata])
alert(Format,Args[,Metadata])
alert(Fun,FunArgs[,Metadata])
Equivalent to log(alert,...).
```

```plaintext
critical(StringOrReport[,Metadata])
critical(Format,Args[,Metadata])
critical(Fun,FunArgs[,Metadata])
Equivalent to log(critical,...).
```

```plaintext
error(StringOrReport[,Metadata])
error(Format,Args[,Metadata])
error(Fun,FunArgs[,Metadata])
Equivalent to log(error,...).
```

```plaintext
warning(StringOrReport[,Metadata])
warning(Format,Args[,Metadata])
warning(Fun,FunArgs[,Metadata])
Equivalent to log(warning,...).
```

```plaintext
notice(StringOrReport[,Metadata])
notice(Format,Args[,Metadata])
notice(Fun,FunArgs[,Metadata])
Equivalent to log(notice,...).
```

```plaintext
info(StringOrReport[,Metadata])
info(Format,Args[,Metadata])
info(Fun,FunArgs[,Metadata])
Equivalent to log(info,...).
```
debug(StringOrReport[,Metadata])
debug(Format,Args[,Metadata])
dump(Fun,FunArgs[,Metadata])

Equivalent to log(debug,...).

log(Level, StringOrReport) -> ok
log(Level, StringOrReport, Metadata) -> ok
log(Level, Format, Args) -> ok
log(Level, Fun, FunArgs) -> ok
log(Level, Format, Args, Metadata) -> ok
log(Level, Fun, FunArgs, Metadata) -> ok

Types:
  Level = level()
  StringOrReport = unicode:chardata() | report()
  Format = io:format()
  Args = [term()]
  Fun = msg_fun()
  FunArgs = term()
  Metadata = metadata()

Log the given message.

Exports

add_handler(HandlerId, Module, Config) -> ok | {error, term()}

Types:
  HandlerId = handler_id()
  Module = module()
  Config = handler_config()

Add a handler with the given configuration.

add_handler_filter(HandlerId, FilterId, Filter) ->
  ok | {error, term()}

Types:
  HandlerId = handler_id()
  FilterId = filter_id()
  Filter = filter()

Add a filter to the specified handler.

The filter fun is called with the log event as the first parameter, and the specified filter_args() as the second parameter.

The return value of the fun specifies if a log event is to be discarded or forwarded to the handler callback:
log_event()

The filter passed. The next handler filter, if any, is applied. If no more filters exist for this handler, the log event is forwarded to the handler callback.

stop

The filter did not pass, and the log event is immediately discarded.

ignore

The filter has no knowledge of the log event. The next handler filter, if any, is applied. If no more filters exist for this handler, the value of the filter_default configuration parameter for the handler specifies if the log event shall be discarded or forwarded to the handler callback.


Some built-in filters exist. These are defined in logger_filters(3).

add_handlers(Application) -> ok | {error, term()}

Types:

    Application = atom()

Reads the application configuration parameter logger and calls add_handlers/1 with its contents.

add_handlers(HandlerConfig) -> ok | {error, term()}

Types:

    HandlerConfig = [config_handler()]
    config_handler() =
        {handler, handler_id(), module(), handler_config()}

This function should be used by custom Logger handlers to make configuration consistent no matter which handler the system uses. Normal usage is to add a call to logger: add_handlers/1 just after the processes that the handler needs are started, and pass the application’s logger configuration as the argument. For example:

```erlang
-behaviour(application).
start(_, []) ->
    case supervisor:start_link({local, my_sup}, my_sup, []) of
        {ok, Pid} ->
            ok = logger:add_handlers(my_app),
            {ok, Pid, []};
        Error -> Error
    end.
```

This reads the logger configuration parameter from the my_app application and starts the configured handlers. The contents of the configuration use the same rules as the logger handler configuration.

If the handler is meant to replace the default handler, the Kernel’s default handler have to be disabled before the new handler is added. A sys.config file that disables the Kernel handler and adds a custom handler could look like this:

```erlang
[[kernel,
  [[logger,
    %% Disable the default Kernel handler
    [{handler, default, undefined}]}}],
  [my_app,
    [[logger,
      %% Enable this handler as the default
      [{handler, default, my_handler, #()}}}]]].
```
add_primary_filter(FilterId, Filter) -> ok | {error, term()}
Types:
  FilterId = filter_id()
  Filter = filter()
Add a primary filter to Logger.
The filter fun is called with the log event as the first parameter, and the specified filter_args() as the second parameter.
The return value of the fun specifies if a log event is to be discarded or forwarded to the handlers:
log_event()
  The filter **passed**, The next primary filter, if any, is applied. If no more primary filters exist, the log event is forwarded to the handler part of Logger, where handler filters are applied.
stop
  The filter **did not pass**, and the log event is immediately discarded.
ignore
  The filter has no knowledge of the log event. The next primary filter, if any, is applied. If no more primary filters exist, the value of the primary filter_default configuration parameter specifies if the log event shall be discarded or forwarded to the handler part.
Some built-in filters exist. These are defined in logger_filters(3).

get_config() ->
  #{
    primary => primary_config(),
    handlers => [handler_config()],
    proxy => olp_config(),
    module_levels =>
      [{module(), level() | all | none}]
  }
Look up all current Logger configuration, including primary, handler, and proxy configuration, and module level settings.

get_handler_config() -> [Config]
Types:
  Config = handler_config()
Look up the current configuration for all handlers.

get_handler_config(HandlerId) -> {ok, Config} | {error, term()}
Types:
  HandlerId = handler_id()
  Config = handler_config()
Look up the current configuration for the given handler.

get_handler_ids() -> [HandlerId]
Types:
HandlerId = handler_id()
Look up the identities for all installed handlers.

get_primary_config() -> Config
Types:
   Config = primary_config()
Look up the current primary configuration for Logger.

get_proxy_config() -> Config
Types:
   Config = olp_config()
Look up the current configuration for the Logger proxy.
For more information about the proxy, see section Logger Proxy in the Kernel User's Guide.

get_module_level() -> [{Module, Level}]
Types:
   Module = module()
   Level = level() | all | none
Look up all current module levels. Returns a list containing one {Module, Level} element for each module for which the module level was previously set with set_module_level/2.

get_module_level(Modules) -> [{Module, Level}]
Types:
   Modules = [Module] | Module
   Module = module()
   Level = level() | all | none
Look up the current level for the given modules. Returns a list containing one {Module, Level} element for each of the given modules for which the module level was previously set with set_module_level/2.

get_process_metadata() -> Meta | undefined
Types:
   Meta = metadata()
Retrieve data set with set_process_metadata/1 or update_process_metadata/1.

i() -> ok
i(What) -> ok
Types:
   What = primary | handlers | proxy | modules | handler_id()
Pretty print the Logger configuration.

remove_handler(HandlerId) -> ok | {error, term()}
Types:
remove_handler_filter(HandlerId, FilterId) -> ok | {error, term()}

Types:
  HandlerId = handler_id()
  FilterId = filter_id()

Remove the filter identified by FilterId from the handler identified by HandlerId.

remove_primary_filter(FilterId) -> ok | {error, term()}

Types:
  FilterId = filter_id()

Remove the primary filter identified by FilterId from Logger.

set_application_level(Application, Level) -> ok | {error, not_loaded}

Types:
  Application = atom()
  Level = level() | all | none

Set the log level for all the modules of the specified application.

This function is a convenience function that calls logger:set_module_level/2 for each module associated with an application.

set_handler_config(HandlerId, Config) -> ok | {error, term()}

Types:
  HandlerId = handler_id()
  Config = handler_config()

Set configuration data for the specified handler. This overwrites the current handler configuration.

To modify the existing configuration, use update_handler_config/2, or, if a more complex merge is needed, read the current configuration with get_handler_config/1, then do the merge before writing the new configuration back with this function.

If a key is removed compared to the current configuration, and the key is known by Logger, the default value is used. If it is a custom key, then it is up to the handler implementation if the value is removed or a default value is inserted.

set_handler_config(HandlerId, Key :: level, Level) -> Return
set_handler_config(HandlerId, Key :: filter_default, FilterDefault) -> Return
set_handler_config(HandlerId, Key :: filters, Filters) -> Return
set_handler_config(HandlerId, Key :: formatter, Formatter) -> Return
set_handler_config(HandlerId, Key :: config, Config) -> Return

Types:
HandlerId = handler_id()
Level = level() | all | none
FilterDefault = log | stop
Filters = [{filter_id(), filter()}]
Formatter = {module(), formatter_config()}
Config = term()
Return = ok | {error, term()}

Add or update configuration data for the specified handler. If the given Key already exists, its associated value will be changed to the given value. If it does not exist, it will be added.

If the value is incomplete, which for example can be the case for the config key, it is up to the handler implementation how the unspecified parts are set. For all handlers in the Kernel application, unspecified data for the config key is set to default values. To update only specified data, and keep the existing configuration for the rest, use update_handler_config/3.

See the definition of the handler_config() type for more information about the different parameters.

set_primary_config(Config) -> ok | {error, term()}
Types:
   Config = primary_config()

Set primary configuration data for Logger. This overwrites the current configuration.
To modify the existing configuration, use update_primary_config/1, or, if a more complex merge is needed, read the current configuration with get_primary_config/0, then do the merge before writing the new configuration back with this function.
If a key is removed compared to the current configuration, the default value is used.

set_primary_config(Key :: level, Level) -> ok | {error, term()}
set_primary_config(Key :: filter_default, FilterDefault) ->
   ok | {error, term()}
set_primary_config(Key :: filters, Filters) ->
   ok | {error, term()}
Types:
   Level = level() | all | none
   FilterDefault = log | stop
   Filters = [{filter_id(), filter()}]

Add or update primary configuration data for Logger. If the given Key already exists, its associated value will be changed to the given value. If it does not exist, it will be added.

set_proxy_config(Config) -> ok | {error, term()}
Types:
   Config = olp_config()

Set configuration data for the Logger proxy. This overwrites the current proxy configuration. Keys that are not specified in the Config map gets default values.
To modify the existing configuration, use update_proxy_config/1, or, if a more complex merge is needed, read the current configuration with get_proxy_config/0, then do the merge before writing the new configuration back with this function.
For more information about the proxy, see section Logger Proxy in the Kernel User's Guide.

```
set_module_level(Modules, Level) -> ok | {error, term()}
```

Types:
- Modules = [module()] | module()
- Level = level() | all | none

Set the log level for the specified modules.

The log level for a module overrides the primary log level of Logger for log events originating from the module in question. Notice, however, that it does not override the level configuration for any handler.

For example: Assume that the primary log level for Logger is `info`, and there is one handler, `h1`, with level `info` and one handler, `h2`, with level `debug`.

With this configuration, no debug messages will be logged, since they are all stopped by the primary log level.

If the level for `mymodule` is now set to `debug`, then debug events from this module will be logged by the handler `h2`, but not by handler `h1`.

Debug events from other modules are still not logged.

To change the primary log level for Logger, use `set_primary_config(level, Level)`.
To change the log level for a handler, use `set_handler_config(HandlerId, level, Level)`.

**Note:**

The originating module for a log event is only detected if the key `mfa` exists in the metadata, and is associated with `{Module, Function, Arity}`. When log macros are used, this association is automatically added to all log events. If an API function is called directly, without using a macro, the logging client must explicitly add this information if module levels shall have any effect.

```
set_process_metadata(Meta) -> ok
```

Types:
- Meta = metadata()

Set metadata which Logger shall automatically insert in all log events produced on the current process.

Location data produced by the log macros, and/or metadata given as argument to the log call (API function or macro), are merged with the process metadata. If the same keys occur, values from the metadata argument to the log call overwrite values from the process metadata, which in turn overwrite values from the location data.

Subsequent calls to this function overwrites previous data set. To update existing data instead of overwriting it, see `update_process_metadata/1`.

```
unset_application_level(Application) -> ok | {error, {not_loaded, Application}}
```

Types:
- Application = atom()

Unset the log level for all the modules of the specified application.

This function is a convinience function that calls `logger:unset_module_level/2` for each module associated with an application.
unset_module_level() -> ok
Remove module specific log settings. After this, the primary log level is used for all modules.

unset_module_level(Modules) -> ok
Types:
  Modules = [module()] | module()
Remove module specific log settings. After this, the primary log level is used for the specified modules.

unset_process_metadata() -> ok
Delete data set with set_process_metadata/1 or update_process_metadata/1.

update_formatter_config(HandlerId, FormatterConfig) ->
  ok | {error, term()}
Types:
  HandlerId = handler_id()
  FormatterConfig = formatter_config()
Update the formatter configuration for the specified handler.
The new configuration is merged with the existing formatter configuration.
To overwrite the existing configuration without any merge, use

```
set_handler_config(HandlerId, formatter,
  {FormatterModule, FormatterConfig}).
```

update_formatter_config(HandlerId, Key, Value) ->
  ok | {error, term()}
Types:
  HandlerId = handler_id()
  Key = atom()
  Value = term()
Update the formatter configuration for the specified handler.
This is equivalent to

```
update_formatter_config(HandlerId, #{Key => Value})
```

update_handler_config(HandlerId, Config) -> ok | {error, term()}
Types:
  HandlerId = handler_id()
  Config = handler_config()
Update configuration data for the specified handler. This function behaves as if it was implemented as follows:

```
{ok, (_, Old)} = logger:get_handler_config(HandlerId),
logger:set_handler_config(HandlerId, maps:merge(Old, Config)).
```
To overwrite the existing configuration without any merge, use set_handler_config/2.
update_handler_config(HandlerId, Key :: level, Level) -> Return
update_handler_config(HandlerId, Key :: filter_default, FilterDefault) -> Return
update_handler_config(HandlerId, Key :: filters, Filters) -> Return
update_handler_config(HandlerId, Key :: formatter, Formatter) -> Return
update_handler_config(HandlerId, Key :: config, Config) -> Return

Types:
  HandlerId = handler_id()
  Level = level() | all | none
  FilterDefault = log | stop
  Filters = [{filter_id(), filter()}]
  Formatter = {module(), formatter_config()}
  Config = term()
  Return = ok | {error, term()}

Add or update configuration data for the specified handler. If the given Key already exists, its associated value will be changed to the given value. If it does not exist, it will be added.

If the value is incomplete, which for example can be the case for the config key, it is up to the handler implementation how the unspecified parts are set. For all handlers in the Kernel application, unspecified data for the config key is not changed. To reset unspecified data to default values, use set_handler_config/3.

See the definition of the handler_config() type for more information about the different parameters.

update_primary_config(Config) -> ok | {error, term()}

Types:
  Config = primary_config()

Update primary configuration data for Logger. This function behaves as if it was implemented as follows:

Old = logger:get_primary_config(),
logger:set_primary_config(maps:merge(Old, Config)).

To overwrite the existing configuration without any merge, use set_primary_config/1.

update_process_metadata(Meta) -> ok

Types:
  Meta = metadata()

Set or update metadata to use when logging from current process

If process metadata exists for the current process, this function behaves as if it was implemented as follows:

logger:set_process_metadata(maps:merge(logger:get_process_metadata(), Meta)).

If no process metadata exists, the function behaves as set_process_metadata/1.

update_proxy_config(Config) -> ok | {error, term()}

Types:
Config = olp_config()
Update configuration data for the Logger proxy. This function behaves as if it was implemented as follows:

```
Old = logger:get_proxy_config(),
logger:set_proxy_config(maps:merge(Old, Config)).
```

To overwrite the existing configuration without any merge, use set_proxy_config/1.

For more information about the proxy, see section Logger Proxy in the Kernel User's Guide.

Exports

**compare_levels**

```
compare_levels(Level1, Level2) -> eq | gt | lt
```
Types:
```
Level1 = Level2 = level() | all | none
```
Compare the severity of two log levels. Returns `gt` if `Level1` is more severe than `Level2`, `lt` if `Level1` is less severe, and `eq` if the levels are equal.

**format_report**

```
format_report(Report) -> FormatArgs
```
Types:
```
Report = report()
FormatArgs = {io:format(), [term()]
```
Convert a log message on report form to `{Format, Args}`. This is the default report callback used by `logger_formatter(3)` when no custom report callback is found. See section Log Message in the Kernel User's Guide for information about report callbacks and valid forms of log messages.

The function produces lines of `Key: Value` from key-value lists. Strings are printed with `~ts` and other terms with `~tp`.

If `Report` is a map, it is converted to a key-value list before formatting as such.

**timestamp**

```
timestamp() -> timestamp()
```
Return a timestamp that can be inserted as the `time` field in the meta data for a log event. It is produced with `os:system_time(microsecond)`.

Notice that Logger automatically inserts a timestamp in the meta data unless it already exists. This function is exported for the rare case when the timestamp must be taken at a different point in time than when the log event is issued.

The following functions are to be exported from a handler callback module.

Exports

**HModule:adding_handler**

```
HModule:adding_handler(Config1) -> {ok, Config2} | {error, Reason}
```
Types:
```
Config1 = Config2 = handler_config()
Reason = term()
```
This callback function is optional.

The function is called on a temporary process when an new handler is about to be added. The purpose is to verify the configuration and initiate all resources needed by the handler.
The handler identity is associated with the \texttt{id} key in \texttt{Config1}.

If everything succeeds, the callback function can add possible default values or internal state values to the configuration, and return the adjusted map in \{\texttt{ok,Config2}\}.

If the configuration is faulty, or if the initiation fails, the callback function must return \{\texttt{error,Reason}\}.

\texttt{HModule:changing_config(SetOrUpdate, OldConfig, NewConfig) -> \{ok, Config\} | \{error, Reason\}}

\textbf{Types:}

\begin{verbatim}
  SetOrUpdate = set | update
  OldConfig = NewConfig = Config = handler_config()
  Reason = term()
\end{verbatim}

This callback function is optional.

The function is called on a temporary process when the configuration for a handler is about to change. The purpose is to verify and act on the new configuration.

\texttt{OldConfig} is the existing configuration and \texttt{NewConfig} is the new configuration.

The handler identity is associated with the \texttt{id} key in \texttt{OldConfig}.

\texttt{SetOrUpdate} has the value \texttt{set} if the configuration change originates from a call to \texttt{set_handler_config/2,3}, and \texttt{update} if it originates from \texttt{update_handler_config/2,3}. The handler can use this parameter to decide how to update the value of the \texttt{config} field, that is, the handler specific configuration data. Typically, if \texttt{SetOrUpdate} equals \texttt{set}, values that are not specified must be given their default values. If \texttt{SetOrUpdate} equals \texttt{update}, the values found in \texttt{OldConfig} must be used instead.

If everything succeeds, the callback function must return a possibly adjusted configuration in \{\texttt{ok,Config}\}.

If the configuration is faulty, the callback function must return \{\texttt{error,Reason}\}.

\texttt{HModule:filter_config(Config) -> FilteredConfig}

\textbf{Types:}

\texttt{Config = FilteredConfig = handler_config()}

This callback function is optional.

The function is called when one of the Logger API functions for fetching the handler configuration is called, for example \texttt{logger:get_handler_config/1}.

It allows the handler to remove internal data fields from its configuration data before it is returned to the caller.

\texttt{HModule:log(LogEvent, Config) -> void()}

\textbf{Types:}

\begin{verbatim}
  LogEvent = log_event()
  Config = handler_config()
\end{verbatim}

This callback function is mandatory.

The function is called when all primary filters and all handler filters for the handler in question have passed for the given log event. It is called on the client process, that is, the process that issued the log event.

The handler identity is associated with the \texttt{id} key in \texttt{Config}.

The handler must log the event.

The return value from this function is ignored by Logger.
HModule:removing_handler(Config) -> ok
Types:

  Config = handler_config()

This callback function is optional.
The function is called on a temporary process when a handler is about to be removed. The purpose is to release all resources used by the handler.
The handler identity is associated with the id key in Config.
The return value is ignored by Logger.
The following functions are to be exported from a formatter callback module.

Exports

FModule:check_config(FConfig) -> ok | {error, Reason}
Types:

  FConfig = formatter_config()
  Reason = term()

This callback function is optional.
The function is called by a Logger when formatter configuration is set or modified. The formatter must validate the given configuration and return ok if it is correct, and {error,Reason} if it is faulty.
The following Logger API functions can trigger this callback:
• logger:add_handler/3
• logger:set_handler_config/2,3
• logger:update_handler_config/2,3
• logger:update_formatter_config/2

See logger_formatter(3) for an example implementation. logger_formatter is the default formatter used by Logger.

FModule:format(LogEvent, FConfig) -> FormattedLogEntry
Types:

  LogEvent = log_event()
  FConfig = formatter_config()
  FormattedLogEntry = unicode:chardata()

This callback function is mandatory.
The function can be called by a log handler to convert a log event term to a printable string. The returned value can, for example, be printed as a log entry to the console or a file using io:put_chars/1,2.
See logger_formatter(3) for an example implementation. logger_formatter is the default formatter used by Logger.

See Also

config(4), erlang(3), io(3), logger_disk_log_h(3), logger_filters(3), logger_formatter(3), logger_std_h(3), unicode(3)
**logger_filters**

Erlang module

All functions exported from this module can be used as primary or handler filters. See `logger:add_primary_filter/2` and `logger:add_handler_filter/3` for more information about how filters are added.

Filters are removed with `logger:remove_primary_filter/1` and `logger:remove_handler_filter/2`.

**Exports**

```erlang
domain(LogEvent, Extra) -> logger:filter_return()
```

Types:

```erlang
LogEvent = logger:log_event()
Extra = {Action, Compare, MatchDomain}
Action = log | stop
Compare = super | sub | equal | not_equal | undefined
MatchDomain = [atom()]
```

This filter provides a way of filtering log events based on a domain field in Metadata. This field is optional, and the purpose of using it is to group log events from, for example, a specific functional area. This allows filtering or other specialized treatment in a Logger handler.

A domain field must be a list of atoms, creating smaller and more specialized domains as the list grows longer. The greatest domain is `[]`, which comprises all possible domains.

For example, consider the following domains:

```erlang
D1 = [otp]
D2 = [otp, sasl]
```

D1 is the greatest of the two, and is said to be a super-domain of D2. D2 is a sub-domain D1. Both D1 and D2 are sub-domains of `[]`.

The above domains are used for logs originating from Erlang/OTP. D1 specifies that the log event comes from Erlang/OTP in general, and D2 indicates that the log event is a so called SASL report.

The `Extra` parameter to the `domain/2` function is specified when adding the filter via `logger:add_primary_filter/2` or `logger:add_handler_filter/3`.

The filter compares the value of the domain field in the log event's metadata (Domain) against MatchDomain. The filter matches if the value of Compare is:

- `sub` and Domain is equal to or a sub-domain of MatchDomain, that is, if MatchDomain is a prefix of Domain.
- `super` and Domain is equal to or a super-domain of MatchDomain, that is, if Domain is a prefix of MatchDomain.
- `equal` and Domain is equal to MatchDomain.
not_equal
    and Domain differs from MatchDomain, or if there is no domain field in metadata.

undefined
    and there is no domain field in metadata. In this case MatchDomain must be set to [].

If the filter matches and Action is log, the log event is allowed. If the filter matches and Action is stop, the log event is stopped.

If the filter does not match, it returns ignore, meaning that other filters, or the value of the configuration parameter filter_default, decide if the event is allowed or not.

Log events that do not contain any domain field, match only when Compare is equal to undefined or not_equal.

Example: stop all events with domain [otp, sasl | _]

```
\logger: add_handler_filter(h1, no_sasl, Filter).
ok
```

level(LogEvent, Extra) -> logger:filter_return()

Types:
    LogEvent = logger:log_event()
    Extra = {Action, Operator, MatchLevel}
    Action = log | stop
    Operator = neq | eq | lt | gt | lteq | gteq
    MatchLevel = logger:level()

This filter provides a way of filtering log events based on the log level. It matches log events by comparing the log level with a specified MatchLevel.

The Extra parameter is specified when adding the filter via logger:add_primary_filter/2 or logger:add_handler_filter/3.

The filter compares the value of the event's log level (Level) to MatchLevel by calling logger:compare_levels(Level, MatchLevel). The filter matches if the value of Operator is:

neq
    and the compare function returns lt or gt.

eq
    and the compare function returns eq.

lt
    and the compare function returns lt.

gt
    and the compare function returns gt.

lteq
    and the compare function returns lt or eq.

gteq
    and the compare function returns gt or eq.
If the filter matches and `Action` is `log`, the log event is allowed. If the filter matches and `Action` is `stop`, the log event is stopped.

If the filter does not match, it returns `ignore`, meaning that other filters, or the value of the configuration parameter `filter_default`, will decide if the event is allowed or not.

Example: only allow debug level log events

```erlang
logger:set_handler_config(h1, filter_default, stop).
Filter = {fun logger_filters:level/2, {log, eq, debug}}.
logger:add_handler_filter(h1, debug_only, Filter).
ok
```

```erlang
progress(LogEvent, Extra) -> logger:filter_return()
```

Types:
- `LogEvent = logger:log_event()`
- `Extra = log | stop`

This filter matches all progress reports from `supervisor` and `application_controller`.

If `Extra` is `log`, the progress reports are allowed. If `Extra` is `stop`, the progress reports are stopped.

The filter returns `ignore` for all other log events.

```erlang
remote_gl(LogEvent, Extra) -> logger:filter_return()
```

Types:
- `LogEvent = logger:log_event()`
- `Extra = log | stop`

This filter matches all events originating from a process that has its group leader on a remote node.

If `Extra` is `log`, the matching events are allowed. If `Extra` is `stop`, the matching events are stopped.

The filter returns `ignore` for all other log events.

See Also

`logger(3)`
logger_formatter

Erlang module

Each Logger handler has a configured formatter specified as a module and a configuration term. The purpose of the formatter is to translate the log events to a final printable string (unicode:chardata()) which can be written to the output device of the handler. See sections Handlers and Formatters in the Kernel User's Guide for more information.

logger_formatter is the default formatter used by Logger.

Data Types

```erlang
config() =
    #{
        chars_limit => integer() >= 1 | unlimited, \
        depth => integer() >= 1 | unlimited, \
        legacy_header => boolean(), \
        max_size => integer() >= 1 | unlimited, \
        report_cb => logger:report_cb(), \
        single_line => boolean(), \
        template => template(), \
        time_designator => byte(), \
        time_offset => integer() | [byte()]
    }
```

The configuration term for logger_formatter is a map, and the following keys can be set as configuration parameters:

- chars_limit = integer() > 0 | unlimited
  
  A positive integer representing the value of the option with the same name to be used when calling io_lib:format/3. This value limits the total number of characters printed for each log event. Notice that this is a soft limit. For a hard truncation limit, see option max_size.
  
  Defaults to unlimited.

- depth = integer() > 0 | unlimited

  A positive integer representing the maximum depth to which terms shall be printed by this formatter. Format strings passed to this formatter are rewritten. The format controls ~p and ~w are replaced with ~P and ~W, respectively, and the value is used as the depth parameter. For details, see io:format/2,3 in STDLIB.

  Defaults to unlimited.

- legacy_header = boolean()

  If set to true a header field is added to logger_formatter's part of Metadata. The value of this field is a string similar to the header created by the old error_logger event handlers. It can be included in the log event by adding the list [logger_formatter,header] to the template. See the description of the template() type for more information.

  Defaults to false.

- max_size = integer() > 0 | unlimited

  A positive integer representing the absolute maximum size a string returned from this formatter can have. If the formatted string is longer, after possibly being limited by chars_limit or depth, it is truncated.

  Defaults to unlimited.
A report callback is used by the formatter to transform log messages on report form to a format string and arguments. The report callback can be specified in the metadata for the log event. If no report callback exists in metadata, `logger_formatter` will use `logger:format_report/1` as default callback.

If this configuration parameter is set, it replaces both the default report callback, and any report callback found in metadata. That is, all reports are converted by this configured function.

**single_line** = boolean()

If set to `true`, each log event is printed as a single line. To achieve this, `logger_formatter` sets the field width to 0 for all `~p` and `~P` control sequences in the format a string (see `io:format/2`), and replaces all newlines in the message with ",", ". White spaces following directly after newlines are removed. Notice that newlines added by the `template` parameter are not replaced.

Defaults to `true`.

**template** = template()

The template describes how the formatted string is composed by combining different data values from the log event. See the description of the `template()` type for more information about this.

**time_designator** = byte()

Timestamps are formatted according to RFC3339, and the time designator is the character used as date and time separator.

Defaults to `$T`.

The value of this parameter is used as the `time_designator` option to `calendar:system_time_to_rfc3339/2`.

**time_offset** = integer() | [byte()]

The time offset, either a string or an integer, to be used when formatting the timestamp.

An empty string is interpreted as local time. The values "Z", "z" or 0 are interpreted as Universal Coordinated Time (UTC).

Strings, other than "Z", "z", or ",", must be on the form ±[hh]:[mm], for example "-02:00" or "+00:00".

Integers must be in microseconds, meaning that the offset 7200000000 is equivalent to "+02:00".

Defaults to an empty string, meaning that timestamps are displayed in local time. However, for backwards compatibility, if the SASL configuration parameter `utc_log=true`, the default is changed to "Z", meaning that timestamps are displayed in UTC.

The value of this parameter is used as the `offset` option to `calendar:system_time_to_rfc3339/2`.

**metakey()** = atom() | [atom()]

**template()** =

```
[metakey() | {metakey(), template(), template()} | string()]
```

The template is a list of atoms, atom lists, tuples and strings. The atoms `level` or `msg`, are treated as placeholders for the severity level and the log message, respectively. Other atoms or atom lists are interpreted as placeholders for metadata, where atoms are expected to match top level keys, and atom lists represent paths to sub keys when the metadata is a nested map. For example the list `[key1,key2]` is replaced by the value of the `key2` field in the nested map below. The atom `key1` on its own is replaced by the complete value of the `key1` field. The values are converted to strings.
Tuples in the template express if-exist tests for metadata keys. For example, the following tuple says that if `key1` exists in the metadata map, print "`key1=Value`", where `Value` is the value that `key1` is associated with in the metadata map. If `key1` does not exist, print nothing.

```
{key1, ["key1=" , key1], []}
```

Strings in the template are printed literally.

The default value for the template configuration parameter depends on the value of the `single_line` and `legacy_header` configuration parameters as follows.

The log event used in the examples is:

```
?LOG_ERROR("name: ~p~nexit_reason: ~p", [my_name, "It crashed"])
```

### legacy_header = true, single_line = false

Default template: `[[logger_formatter, header], "\n", msg, "\n"]`

Example log entry:

```
ERROR REPORT==== 17-May-2018::18:30:19.453447 ===
name: my_name
exit_reason: "It crashed"
```

Notice that all eight levels can occur in the heading, not only ERROR, WARNING or INFO as error_logger produces. And microseconds are added at the end of the timestamp.

### legacy_header = true, single_line = true

Default template: `[[logger_formatter, header], "\n", msg, "\n"]`

Notice that the template is here the same as for single_line=false, but the resulting log entry differs in that there is only one line after the heading:

```
ERROR REPORT==== 17-May-2018::18:31:06.952665 ===
name: my_name, exit_reason: "It crashed"
```

### legacy_header = false, single_line = true

Default template: `[time, " ", level, " : ", msg, "\n"]`

Example log entry:

```
2018-05-17T18:31:31.152864+02:00 error: name: my_name, exit_reason: "It crashed"
```

### legacy_header = false, single_line = false

Default template: `[time, " ", level, ":\n", msg, "\n"]`

Example log entry:

```
2018-05-17T18:32:20.105422+02:00 error: name: my_name
exit_reason: "It crashed"
```
Exports

check_config(Config) -> ok | {error, term()}
Types:
  Config = config()
The function is called by Logger when the formatter configuration for a handler is set or modified. It returns ok if the configuration is valid, and {error,term()} if it is faulty.

The following Logger API functions can trigger this callback:
  • logger:add_handler/3
  • logger:set_handler_config/2,3
  • logger:update_handler_config/2
  • logger:update_formatter_config/2

format(LogEvent, Config) -> unicode:chardata()
Types:
  LogEvent = logger:log_event()
  Config = config()
This the formatter callback function to be called from handlers. The log event is processed as follows:
  • If the message is on report form, it is converted to {Format,Args} by calling the report callback. See section Log Message in the Kernel User's Guide for more information about report callbacks and valid forms of log messages.
  • The message size is limited according to the values of configuration parameters chars_limit and depth.
  • The full log entry is composed according to the template.
  • If the final string is too long, it is truncated according to the value of configuration parameter max_size.

See Also

calendar(3), error_logger(3), io(3), io_lib(3), logger(3), maps(3), sasl(6), unicode(3)
This is the standard handler for Logger. Multiple instances of this handler can be added to Logger, and each instance
prints logs to standard_io, standard_error, or to file.

The handler has an overload protection mechanism that keeps the handler process and the Kernel application alive
during high loads of log events. How overload protection works, and how to configure it, is described in the User’s
Guide.

To add a new instance of the standard handler, use `logger:add_handler/3`. The handler configuration argument
is a map which can contain general configuration parameters, as documented in the User’s Guide, and handler
specific parameters. The specific data is stored in a sub map with the key `config`, and can contain the following
parameters:

```erlang
{type, standard_io | standard_error | file | {device, io:device()}}
```

Specifies the log destination.

The value is set when the handler is added, and it cannot be changed in runtime.

Defaults to `standard_io`, unless parameter `file` is given, in which case it defaults to `file`.

```erlang
{file, file:filename()}
```

This specifies the name of the log file when the handler is of type `file`.

The value is set when the handler is added, and it cannot be changed in runtime.

Defaults to the same name as the handler identity, in the current directory.

```erlang
{modes, [file:mode()]}[[1]]
```

This specifies the file modes to use when opening the log file, see `file:open/2`. If `modes` are not specified,
the default list used is `[raw, append, delayed_write]`. If `modes` are specified, the list replaces the default
modes list with the following adjustments:

- If `raw` is not found in the list, it is added.
- If none of `write`, `append` or `exclusive` is found in the list, `append` is added.
- If none of `delayed_write` or `{delayed_write, Size, Delay}` is found in the list, `delayed_write` is added.

Log files are always UTF-8 encoded. The encoding cannot be changed by setting the mode `{encoding, Encoding}`.

The value is set when the handler is added, and it cannot be changed in runtime.

Defaults to `[raw, append, delayed_write]`.

```erlang
{max_no_bytes, pos_integer() | infinity}
```

This parameter specifies if the log file should be rotated or not. The value `infinity` means the log file will
grow indefinitely, while an integer value specifies at which file size (bytes) the file is rotated.

Defaults to `infinity`.

```erlang
{max_no_files, non_neg_integer()}
```

This parameter specifies the number of rotated log file archives to keep. This has meaning only if
`max_no_bytes` is set to an integer value.
The log archives are named FileName.0, FileName.1, ..., FileName.N, where FileName is the name of the current log file. FileName.0 is the newest of the archives. The maximum value for N is the value of max_no_files minus 1.

Notice that setting this value to 0 does not turn off rotation. It only specifies that no archives are kept.

Defaults to 0.

compress_on_rotate = boolean()

This parameter specifies if the rotated log file archives shall be compressed or not. If set to true, all archives are compressed with gzip, and renamed to FileName.N.gz

compress_on_rotate has no meaning if max_no_bytes has the value infinity.

Defaults to false.

file_check = non_neg_integer()

When logger_std_h logs to a file, it reads the file information of the log file prior to each write operation. This is to make sure the file still exists and has the same inode as when it was opened. This implies some performance loss, but ensures that no log events are lost in the case when the file has been removed or renamed by an external actor.

In order to allow minimizing the performance loss, the file_check parameter can be set to a positive integer value, N. The handler will then skip reading the file information prior to writing, as long as no more than N milliseconds have passed since it was last read.

Notice that the risk of losing log events grows when the file_check value grows.

Defaults to 0.

filesync_repeat_interval = pos_integer() | no_repeat

This value, in milliseconds, specifies how often the handler does a file sync operation to write buffered data to disk. The handler attempts the operation repeatedly, but only performs a new sync if something has actually been logged.

If no_repeat is set as value, the repeated file sync operation is disabled, and it is the operating system settings that determine how quickly or slowly data is written to disk. The user can also call the filesync/1 function to perform a file sync.

Defaults to 5000 milliseconds.

Other configuration parameters exist, to be used for customizing the overload protection behaviour. The same parameters are used both in the standard handler and the disk_log handler, and are documented in the User\'s Guide.

Notice that if changing the configuration of the handler in runtime, the type, file, or modes parameters must not be modified.

Example of adding a standard handler:

```erlang
logger:add_handler(my_standard_h, logger_std_h, #{config => #{file => ".system_info.log", filesync_repeat_interval => 1000}}).
```

To set the default handler, that starts initially with the Kernel application, to log to file instead of standard_io, change the Kernel default logger configuration. Example:

```erlang
erl -kernel logger '[[handler,default,logger_std_h, #{config => #{file => ".log.log"}}]']
```

An example of how to replace the standard handler with a disk_log handler at startup is found in the logger_disk_log_h manual.
Exports

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

Types:

```plaintext
Name = atom()
Reason = handler_busy | {badarg, term()}
```

Write buffered data to disk.

See Also

`logger(3), logger_disk_log_h(3)`
This is a handler for Logger that offers circular (wrapped) logs by using disk_log. Multiple instances of this handler can be added to Logger, and each instance prints to its own disk log file, created with the name and settings specified in the handler configuration.

The default standard handler, logger_std_h, can be replaced by a disk_log handler at startup of the Kernel application. See an example of this below.

The handler has an overload protection mechanism that keeps the handler process and the Kernel application alive during high loads of log events. How overload protection works, and how to configure it, is described in the User's Guide.

To add a new instance of the disk_log handler, use logger:add_handler/3. The handler configuration argument is a map which can contain general configuration parameters, as documented in the User's Guide, and handler specific parameters. The specific data is stored in a sub map with the key config, and can contain the following parameters:

**file**
- This is the full name of the disk log file. The option corresponds to the name property in the dlog_option() datatype.
- The value is set when the handler is added, and it cannot be changed in runtime.
-Defaults to the same name as the handler identity, in the current directory.

**type**
- This is the disk log type, wrap or halt. The option corresponds to the type property in the dlog_option() datatype.
- The value is set when the handler is added, and it cannot be changed in runtime.
- Defaults to wrap.

**max_no_files**
- This is the maximum number of files that disk_log uses for its circular logging. The option corresponds to the MaxNoFiles element in the size property in the dlog_option() datatype.
- The value is set when the handler is added, and it cannot be changed in runtime.
- Defaults to 10.
- The setting has no effect on a halt log.

**max_no_bytes**
- This is the maximum number of bytes that is written to a log file before disk_log proceeds with the next file in order, or generates an error in case of a full halt log. The option corresponds to the MaxNoBytes element in the size property in the dlog_option() datatype.
- The value is set when the handler is added, and it cannot be changed in runtime.
- Defaults to 1048576 bytes for a wrap log, and infinity for a halt log.

**filesync_repeat_interval**
- This value, in milliseconds, specifies how often the handler does a disk_log sync operation to write buffered data to disk. The handler attempts the operation repeatedly, but only performs a new sync if something has actually been logged.
Defaults to 5000 milliseconds.

If no_repeat is set as value, the repeated sync operation is disabled. The user can also call the \texttt{filesync/1} function to perform a disk_log sync.

Other configuration parameters exist, to be used for customizing the overload protection behaviour. The same parameters are used both in the standard handler and the disk_log handler, and are documented in the User's Guide.

Notice that when changing the configuration of the handler in runtime, the disk_log options (file, type, max_no_files, max_no_bytes) must not be modified.

Example of adding a disk_log handler:

\begin{verbatim}
logger:add_handler(my_disk_log_h, logger_disk_log_h,
                   #{config => #{file => "/my_disk_log",
                                 type => wrap,
                                 max_no_files => 4,
                                 max_no_bytes => 10000,
                                 syscall_repeat_interval => 1000}}).
\end{verbatim}

To use the disk_log handler instead of the default standard handler when starting an Erlang node, change the Kernel default logger to use logger_disk_log_h. Example:

\begin{verbatim}
erl -kernel logger '[[handler,default,logger_disk_log_h,
                        #{config => #{file => "/system_disk_log"}}]]'
\end{verbatim}

Exports

\begin{verbatim}
filesync(Name) -> ok | {error, Reason}
\end{verbatim}

Types:

\begin{verbatim}
Name = atom()
Reason = handler_busy | {badarg, term()}
\end{verbatim}

Write buffered data to disk.

See Also

logger(3), logger_std_h(3), disk_log(3)
This module provides an API for the network interface.

**Note:**
There is currently no support for Windows.

**Note:**
The content of this file is only valid if the system has been built with 'socket' (esock) support, which is the default.

### Data Types

**address_info()** =

```erlang
#{family := term(),
 socktype := term(),
 protocol := term(),
 address := term()}
```

**ifaddrs()** =

```erlang
#{name := string(),
 flags := [ifaddrs_flag()],
 addr := socket:sockaddr(),
 netmask := socket:sockaddr(),
 broadaddr := socket:sockaddr(),
 dstaddr := socket:sockaddr()}
```

This type defines all addresses (and flags) associated with the interface.

Not all fields of this map has to be present. The flags field can be used to test for some of the fields. For example `broadaddr` will only be present if the `broadcast` flag is present in `flags`.

**ifaddrs_flag()** =

- `up`
- `broadcast`
- `debug`
- `loopback`
- `pointopoint`
- `notrailers`
- `running`
- `noarp`
- `promisc`
- `master`
- `slave`
- `multicast`
- `portsel`
- `automedia`
- `dynamic`

**ifaddrs_filter()** =

- `all`
- `default`
- `inet`
- `inet6`
- `packet`

**ifaddrs_filter_map()**

**ifaddrs_filter_fun()**

- `all`
  - All interfaces
- `default`
  - Interfaces with address family `inet` and `inet6`
  - Interfaces with only the specified address family

**ifaddrs_filter_map()** =

```erlang
#{family := default | inet | inet6 | packet | all,
```
flags := any | [ifaddrs_flag()]

The family field can only have the (above) specified values (and not all the values of socket:domain()).

The use of the flags field is that any flag provided must exist for the interface.

For example, if family is set to inet and flags to [broadcast, multicast] only interfaces with address family inet and the flags broadcast and multicast will be listed.

ifaddrs_filter_fun() = fun((ifaddrs()) -> boolean())

For each ifaddrs entry, return either true to keep the entry or false to discard the entry.

For example, to get an interface list which only contains non-loopback inet interfaces:

```erlang
net:getifaddrs(fun(#{addr := #{family := inet},
                  flags := Flags}) ->
    not lists:member(loopback, Flags);
    (_) ->
        false
end).
```

name_info() = #{host := string(), service := string()}

name_info_flags() = [name_info_flag() | name_info_flag_ext()]

name_info_flag() = namereqd | dgram | nofqdn | numerichost | numericserv

name_info_flag_ext() = idn

network_interface_name() = string()

network_interface_index() = integer() >= 0

Exports

gethostname() -> {ok, HostName} | {error, Reason}

Types:
    HostName = string()
    Reason = term()

Returns the name of the current host.

getnameinfo(SockAddr) -> {ok, Info} | {error, Reason}
getnameinfo(SockAddr, Flags) -> {ok, Info} | {error, Reason}

Types:
    SockAddr = term()
    Flags = name_info_flags() | undefined
    Info = name_info()
    Reason = term()

Address-to-name translation in a protocol-independent manner.

This function is the inverse of getaddrinfo. It converts a socket address to a corresponding host and service.

getaddrinfo(Host) -> {ok, Info} | {error, Reason}
getaddrinfo(Host, Service :: undefined) ->
    {ok, Info} | {error, Reason}
getaddrinfo(Host :: undefined, Service) ->
getaddrinfo(Host, Service) -> {ok, Info} | {error, Reason}
Types:
  Host = Service = string()
  Info = [address_info()]
  Reason = term()
Network address and service translation.
This function is the inverse of getnameinfo. It converts host and service to a corresponding socket address.
One of the Host and Service may be undefined but not both.

getifaddrs() -> {ok, IfAddrs} | {error, Reason}
getifaddrs(Filter) -> {ok, IfAddrs} | {error, Reason}
getifaddrs(Namespace) -> {ok, IfAddrs} | {error, Reason}
getifaddrs(Filter, Namespace) -> {ok, IfAddrs} | {error, Reason}
Types:
  Filter = ifaddrs_filter()
  Namespace = file:filename_all()
  IfAddrs = [ifaddrs()]
  Reason = term()
Get interface addresses.
This function is used to get the machines interface addresses, possibly filtered according to Filter.
By default, a filter with the content: #{family => default, flags => any} is used. This will return all interfaces with addresses in the inet and inet6 families.

if_name2index(Name) -> {ok, Idx} | {error, Reason}
Types:
  Name = network_interface_name()
  Idx = network_interface_index()
  Reason = term()
Mappings between network interface names and indexes.

if_index2name(Idx) -> {ok, Name} | {error, Reason}
Types:
  Idx = network_interface_index()
  Name = network_interface_name()
  Reason = term()
Mappings between network interface index and names.

if_names() -> Names | {error, Reason}
Types:
Names = [{Idx, If}]
Idx = network_interface_index()
If = network_interface_name()
Reason = term()

Get network interface names and indexes.
This module contains various network utility functions.

Exports

dns_hostname(Host) -> {ok, Name} | {error, Host}

Types:
  Host = atom() | string()
  Name = string()

Returns the official name of Host, or {error, Host} if no such name is found. See also inet(3).

host_file() -> Hosts | {error, Reason}

Types:
  Hosts = [Host :: atom()]
  Reason =
    file:posix() | badarg | terminated | system_limit | {Line :: integer(), Mod :: module(), Term :: term()}

Reads file .hosts.erlang, see section Files. Returns the hosts in this file as a list. Returns {error, Reason} if the file cannot be read or the Erlang terms on the file cannot be interpreted.

localhost() -> Name

Types:
  Name = string()

Returns the name of the local host. If Erlang was started with command-line flag -name, Name is the fully qualified name.

names() -> {ok, [{Name, Port}]} | {error, Reason}
names(Host) -> {ok, [{Name, Port}]} | {error, Reason}

Types:
  Host = atom() | string() | inet:ip_address()
  Name = string()
  Port = integer() >= 0
  Reason = address | file:posix()

Similar to epmd -names, see erts:epmd(1). Host defaults to the local host. Returns the names and associated port numbers of the Erlang nodes that epmd registered at the specified host. Returns {error, address} if epmd is not operational.

Example:

(arnen@dunn)1> net_adm:names().
{ok, [{"arnen",40262}]}
ping(Node) -> pong | pang

Types:
    Node = atom()

Sets up a connection to Node. Returns pong if it is successful, otherwise pang.

world() -> [node()]
world(Arg) -> [node()]

Types:
    Arg = verbosity()
    verbosity() = silent | verbose

Calls names(Host) for all hosts that are specified in the Erlang host file .hosts.erlang, collects the replies, and then evaluates ping(Node) on all those nodes. Returns the list of all nodes that are successfully pinged.

Arg defaults to silent. If Arg == verbose, the function writes information about which nodes it is pinging to stdout.

This function can be useful when a node is started, and the names of the other network nodes are not initially known.

Returns {error, Reason} if host_file() returns {error, Reason}.

world_list(Hosts) -> [node()]
world_list(Hosts, Arg) -> [node()]

Types:
    Hosts = [atom()]
    Arg = verbosity()
    verbosity() = silent | verbose

Same as world/0,1, but the hosts are specified as argument instead of being read from .hosts.erlang.

Files

File .hosts.erlang consists of a number of host names written as Erlang terms. It is looked for in the current work directory, the user's home directory, and $OTP_ROOT (the root directory of Erlang/OTP), in that order.

The format of file .hosts.erlang must be one host name per line. The host names must be within quotes.

Example:

'super.eua.ericsson.se'.
'renat.eua.ericsson.se'.
'grouse.eua.ericsson.se'.
'gauffin1.eua.ericsson.se'.
^ (new line)
The net kernel is a system process, registered as `net_kernel`, which must be operational for distributed Erlang to work. The purpose of this process is to implement parts of the BIFs `spawn/4` and `spawn_link/4`, and to provide monitoring of the network.

An Erlang node is started using command-line flag `-name` or `-sname`:

```bash
$ erl -sname foobar
```

It is also possible to call `net_kernel:start([foobar])` directly from the normal Erlang shell prompt:

```erlang
1> net_kernel:start([foobar, shortnames]).
{ok,<0.64.0>}
(foobar@gringotts)2>
```

If the node is started with command-line flag `-sname`, the node name is `foobar@Host`, where `Host` is the short name of the host (not the fully qualified domain name). If started with flag `-name`, the node name is `foobar@Host`, where `Host` is the fully qualified domain name. For more information, see `erl`.

Normally, connections are established automatically when another node is referenced. This functionality can be disabled by setting Kernel configuration parameter `dist_auto_connect` to `never`, see `kernel(6)`. In this case, connections must be established explicitly by calling `connect_node/1`.

Which nodes that are allowed to communicate with each other is handled by the magic cookie system, see section Distributed Erlang in the Erlang Reference Manual.

### Warning:

Starting a distributed node without also specifying `-proto_dist inet_tls` will expose the node to attacks that may give the attacker complete access to the node and in extension the cluster. When using un-secure distributed nodes, make sure that the network is configured to keep potential attackers out. See the Using SSL for Erlang Distribution User's Guide for details on how to setup a secure distributed node.

### Exports

`allow(Nodes) -> ok | error`

Types:

- `Nodes` = `[node()]`

Permits access to the specified set of nodes.

Before the first call to `allow/1`, any node with the correct cookie can be connected. When `allow/1` is called, a list of allowed nodes is established. Any access attempts made from (or to) nodes not in that list will be rejected.

Subsequent calls to `allow/1` will add the specified nodes to the list of allowed nodes. It is not possible to remove nodes from the list.

Returns `error` if any element in `Nodes` is not an atom.
connect_node(Node) -> boolean() | ignored
Types:
    Node = node()
Establishes a connection to Node. Returns true if a connection was established or was already established or if Node is the local node itself. Returns false if the connection attempt failed, and ignored if the local node is not alive.

get_net_ticktime() -> Res
Types:
    Res = NetTicktime | {ongoing_change_to, NetTicktime} | ignored
    NetTicktime = integer() >= 1
Gets net_ticktime (see kernel(6)).
Defined return values (Res):
NetTicktime
    net_ticktime is NetTicktime seconds.
{ongoing_change_to, NetTicktime}
    net_kernel is currently changing net_ticktime to NetTicktime seconds.
ignored
    The local node is not alive.

getopts(Node, Options) ->
    {ok, OptionValues} | {error, Reason} | ignored
Types:
    Node = node()
    Options = [inet:socket_getopt()]
    OptionValues = [inet:socket_setopt()]
    Reason = inet:posix() | noconnection
Get one or more options for the distribution socket connected to Node.
If Node is a connected node the return value is the same as from inet:getopts(Socket, Options) where Socket is the distribution socket for Node.
Returns ignored if the local node is not alive or {error, noconnection} if Node is not connected.

monitor_nodes(Flag) -> ok | Error
monitor_nodes(Flag, Options) -> ok | Error
Types:
    Flag = boolean()
    Options = [Option]
    Option = {node_type, NodeType} | nodedown_reason
    NodeType = visible | hidden | all
    Error = error | {error, term()}
The calling process subscribes or unsubscribes to node status change messages. A nodeup message is delivered to all subscribing processes when a new node is connected, and a nodedown message is delivered when a node is disconnected.
If `Flag` is true, a new subscription is started. If `Flag` is false, all previous subscriptions started with the same `Options` are stopped. Two option lists are considered the same if they contain the same set of options.

As from Kernel version 2.11.4, and ERTS version 5.5.4, the following is guaranteed:

- nodeup messages are delivered before delivery of any message from the remote node passed through the newly established connection.
- nodedown messages are not delivered until all messages from the remote node that have been passed through the connection have been delivered.

Notice that this is **not** guaranteed for Kernel versions before 2.11.4.

As from Kernel version 2.11.4, subscriptions can also be made before the `net_kernel` server is started, that is, `net_kernel:monitor_nodes/[1,2]` does not return ignored.

As from Kernel version 2.13, and ERTS version 5.7, the following is guaranteed:

- nodeup messages are delivered after the corresponding node appears in results from `erlang:nodes/X`.
- nodedown messages are delivered after the corresponding node has disappeared in results from `erlang:nodes/X`.

Notice that this is **not** guaranteed for Kernel versions before 2.13.

The format of the node status change messages depends on `Options`. If `Options` is `[]`, which is the default, the format is as follows:

```
{nodeup, Node} | {nodedown, Node}
Node = node()
```

If `Options` is not `[]`, the format is as follows:

```
{nodeup, Node, InfoList} | {nodedown, Node, InfoList}
Node = node()
InfoList = [{Tag, Val}]
```

`InfoList` is a list of tuples. Its contents depends on `Options`, see below.

Also, when `Options == []`, only visible nodes, that is, nodes that appear in the result of `erlang:nodes/0`, are monitored.

`Option` can be any of the following:

```
{node_type, NodeType}
Valid values for NodeType:
visible
  Subscribe to node status change messages for visible nodes only. The tuple `node_type, visible` is included in `InfoList`.
hidden
  Subscribe to node status change messages for hidden nodes only. The tuple `node_type, hidden` is included in `InfoList`.
all
  Subscribe to node status change messages for both visible and hidden nodes. The tuple `node_type, visible | hidden` is included in `InfoList`.
```

`nodedown_reason`

The tuple `nodedown_reason, Reason` is included in `InfoList` in nodedown messages.
Reason can, depending on which distribution module or process that is used be any term, but for the standard TCP distribution module it is any of the following:

- **connection_setup_failed**
  - The connection setup failed (after nodeup messages were sent).

- **no_network**
  - No network is available.

- **net_kernel_terminated**
  - The net_kernel process terminated.

- **shutdown**
  - Unspecified connection shutdown.

- **connection_closed**
  - The connection was closed.

- **disconnect**
  - The connection was disconnected (forced from the current node).

- **net_tick_timeout**
  - Net tick time-out.

- **send_net_tick_failed**
  - Failed to send net tick over the connection.

- **get_status_failed**
  - Status information retrieval from the Port holding the connection failed.

```erlang
set_net_ticktime(NetTicktime) -> Res
set_net_ticktime(NetTicktime, TransitionPeriod) -> Res
```

Types:

- **NetTicktime** = integer() >= 1
- **TransitionPeriod** = integer() >= 0
- **Res** = unchanged | change_initiated | {ongoing_change_to, NewNetTicktime}
- **NewNetTicktime** = integer() >= 1

Sets net_ticktime (see kernel(6)) to NetTicktime seconds. TransitionPeriod defaults to 60.

Some definitions:

- **Minimum transition traffic interval (MTTI)**
  
  ```erlang
  minimum(NetTicktime, PreviousNetTicktime)*1000 div 4 milliseconds.
  ```

- **Transition period**
  
  The time of the least number of consecutive MTTIs to cover TransitionPeriod seconds following the call to set_net_ticktime/2 (that is, ((TransitionPeriod*1000 - 1) div MTTI + 1)*MTTI milliseconds).
If $\text{NetTicktime} < \text{PreviousNetTicktime}$, the $\text{net\_ticktime}$ change is done at the end of the transition period; otherwise at the beginning. During the transition period, $\text{net\_kernel}$ ensures that there is outgoing traffic on all connections at least every MTTI millisecond.

**Note:**

The $\text{net\_ticktime}$ changes must be initiated on all nodes in the network (with the same $\text{NetTicktime}$) before the end of any transition period on any node; otherwise connections can erroneously be disconnected.

Returns one of the following:

- **unchanged**
  - $\text{net\_ticktime}$ already has the value of $\text{NetTicktime}$ and is left unchanged.

- **change_initiated**
  - $\text{net\_kernel}$ initiated the change of $\text{net\_ticktime}$ to $\text{NetTicktime}$ seconds.

- **{ongoing_change_to, NewNetTicktime}**
  - The request is **ignored** because $\text{net\_kernel}$ is busy changing $\text{net\_ticktime}$ to $\text{NewNetTicktime}$ seconds.

### setopts(Node, Options) -> ok | {error, Reason} | ignored

**Types:**

- **Node** = node() | new
- **Options** = [inet:socket_setopt()]
- **Reason** = inet:posix() | noconnection

Set one or more options for distribution sockets. Argument Node can be either one node name or the atom new to affect the distribution sockets of all future connected nodes.

The return value is the same as from inet:setopts/2 or {error, noconnection} if Node is not a connected node or new.

If Node is new the Options will then also be added to kernel configuration parameters inet_dist_listen_options and inet_dist_connect_options.

Returns ignored if the local node is not alive.

### start([Name]) -> {ok, pid()} | {error, Reason}

### start([Name, NameType]) -> {ok, pid()} | {error, Reason}

### start([Name, NameType, Ticktime]) -> {ok, pid()} | {error, Reason}

**Types:**

- **Name** = atom()
- **NameType** = shortnames | longnames
- **Reason** = {already_started, pid()} | term()

Turns a non-distributed node into a distributed node by starting $\text{net\_kernel}$ and other necessary processes.

Notice that the argument is a list with exactly one, two, or three arguments. NameType defaults to longnames and Ticktime to 15000.

### stop() -> ok | {error, Reason}

**Types:**
Reason = not_allowed | not_found

Turns a distributed node into a non-distributed node. For other nodes in the network, this is the same as the node going down. Only possible when the net kernel was started using start/1, otherwise {error, not_allowed} is returned. Returns {error, not_found} if the local node is not alive.
The functions in this module are operating system-specific. Careless use of these functions results in programs that will only run on a specific platform. On the other hand, with careful use, these functions can be of help in enabling a program to run on most platforms.

**Note:**

File operations used to accept filenames containing null characters (integer value zero). This caused the name to be truncated and in some cases arguments to primitive operations to be mixed up. Filenames containing null characters inside the filename are now rejected and will cause primitive file operations to fail.

Also environment variable operations used to accept names and values of environment variables containing null characters (integer value zero). This caused operations to silently produce erroneous results. Environment variable names and values containing null characters inside the name or value are now rejected and will cause environment variable operations to fail.

### Data Types

**env_var_name() = nonempty_string()**

A string containing valid characters on the specific OS for environment variable names using file:native_name_encoding() encoding. Note that specifically null characters (integer value zero) and $=$ characters are not allowed. However, note that not all invalid characters necessarily will cause the primitive operations to fail, but may instead produce invalid results.

**env_var_value() = string()**

A string containing valid characters on the specific OS for environment variable values using file:native_name_encoding() encoding. Note that specifically null characters (integer value zero) are not allowed. However, note that not all invalid characters necessarily will cause the primitive operations to fail, but may instead produce invalid results.

**env_var_name_value() = nonempty_string()**

Assuming that environment variables has been correctly set, a strings containing valid characters on the specific OS for environment variable names and values using file:native_name_encoding() encoding. The first $=$ characters appearing in the string separates environment variable name (on the left) from environment variable value (on the right).

**os_command() = atom() | io_lib:chars()**

All characters needs to be valid characters on the specific OS using file:native_name_encoding() encoding. Note that specifically null characters (integer value zero) are not allowed. However, note that not all invalid characters not necessarily will cause os:cmd/1 to fail, but may instead produce invalid results.

**os_command_opts() = #{max_size => integer() >= 0 | infinity}**

Options for os:cmd/2

max_size

The maximum size of the data returned by the os:cmd call. See the os:cmd/2 documentation for more details.
Exports

`cmd(Command) -> string()`  
`cmd(Command, Options) -> string()`

Types:  
  `Command = os_command()`  
  `Options = os_command_opts()`

Executes `Command` in a command shell of the target OS, captures the standard output of the command, and returns this result as a string.

**Warning:**

Previous implementation used to allow all characters as long as they were integer values greater than or equal to zero. This sometimes lead to unwanted results since null characters (integer value zero) often are interpreted as string termination. The current implementation rejects these.

Examples:

```
LsOut = os:cmd("ls"), % on unix platform
DirOut = os:cmd("dir"), % on Win32 platform
```

Notice that in some cases, standard output of a command when called from another program (for example, `os:cmd/1`) can differ, compared with the standard output of the command when called directly from an OS command shell.

`os:cmd/2` was added in kernel-5.5 (OTP-20.2.1). It makes it possible to pass an options map as the second argument in order to control the behaviour of `os:cmd`. The possible options are:

- `max_size`  
  The maximum size of the data returned by the `os:cmd` call. This option is a safety feature that should be used when the command executed can return a very large, possibly infinite, result.

```
> os:cmd("cat /dev/zero", #{max_size => 20}).
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
```

`find_executable(Name) -> Filename | false`  
`find_executable(Name, Path) -> Filename | false`

Types:  
  `Name = Path = Filename = string()`

These two functions look up an executable program, with the specified name and a search path, in the same way as the underlying OS. `find_executable/1` uses the current execution path (that is, the environment variable `PATH` on Unix and Windows).

`Path`, if specified, is to conform to the syntax of execution paths on the OS. Returns the absolute filename of the executable program `Name`, or `false` if the program is not found.

`getenv() -> [env_var_name_value()]`

Returns a list of all environment variables. Each environment variable is expressed as a single string on the format "VarName=Value", where `VarName` is the name of the variable and `Value` its value.

If Unicode filename encoding is in effect (see the `erl` manual page), the strings can contain characters with codepoints > 255.
getenv(VarName) -> Value | false

Types:

VarName = env_var_name()
Value = env_var_value()

Returns the Value of the environment variable VarName, or false if the environment variable is undefined.

If Unicode filename encoding is in effect (see the erl manual page), the strings VarName and Value can contain characters with codepoints > 255.

getenv(VarName, DefaultValue) -> Value

Types:

VarName = env_var_name()
DefaultValue = Value = env_var_value()

Returns the Value of the environment variable VarName, or DefaultValue if the environment variable is undefined.

If Unicode filename encoding is in effect (see the erl manual page), the strings VarName and Value can contain characters with codepoints > 255.

getpid() -> Value

Types:

Value = string()

Returns the process identifier of the current Erlang emulator in the format most commonly used by the OS environment. Returns Value as a string containing the (usually) numerical identifier for a process. On Unix, this is typically the return value of the getpid() system call. On Windows, the process id as returned by the GetCurrentProcessId() system call is used.

putenv(VarName, Value) -> true

Types:

VarName = env_var_name()
Value = env_var_value()

Sets a new Value for environment variable VarName.

If Unicode filename encoding is in effect (see the erl manual page), the strings VarName and Value can contain characters with codepoints > 255.

On Unix platforms, the environment is set using UTF-8 encoding if Unicode filename translation is in effect. On Windows, the environment is set using wide character interfaces.

Note:

VarName is not allowed to contain an $= character. Previous implementations used to just let the $= character through which silently caused erroneous results. Current implementation will instead throw a badarg exception.

set_signal(Signal, Option) -> ok

Types:

Signal =
    sighup | sigquit | sigabrt | sigalrm | sigterm | sigusr1 |
Enables or disables OS signals.
Each signal may be set to one of the following options:

- **ignore**
  - This signal will be ignored.
- **default**
  - This signal will use the default signal handler for the operating system.
- **handle**
  - This signal will notify `erl_signal_server` when it is received by the Erlang runtime system.

**system_time() -> integer()**
Returns the current OS system time in native time unit.

<table>
<thead>
<tr>
<th>Note:</th>
</tr>
</thead>
<tbody>
<tr>
<td>This time is <strong>not</strong> a monotonically increasing time.</td>
</tr>
</tbody>
</table>

**system_time(Unit) -> integer()**
Types:

- **Unit** = `erlang:time_unit()`

Returns the current OS system time converted into the `Unit` passed as argument.
Calling `os:system_time(Unit)` is equivalent to `erlang:convert_time_unit(os:system_time(), native, Unit)`.

<table>
<thead>
<tr>
<th>Note:</th>
</tr>
</thead>
<tbody>
<tr>
<td>This time is <strong>not</strong> a monotonically increasing time.</td>
</tr>
</tbody>
</table>

**timestamp() -> Timestamp**
Types:

- **Timestamp** = `erlang:timestamp()`
- **Timestamp** = `{MegaSecs, Secs, MicroSecs}`

Returns the current OS system time in the same format as `erlang:timestamp/0`. The tuple can be used together with function `calendar:now_to_universal_time/1` or `calendar:now_to_local_time/1` to get calendar time. Using the calendar time, together with the `MicroSecs` part of the return tuple from this function, allows you to log time stamps in high resolution and consistent with the time in the rest of the OS.

Example of code formatting a string in format "DD Mon YYYY HH:MM:SS.mmmmmm", where DD is the day of month, Mon is the textual month name, YYYY is the year, HH:MM:SS is the time, and mmmmmm is the microseconds in six positions:
This module can be used as follows:

```
1> io:format("~s~n",[print_time:format_utc_timestamp()]).
29 Apr 2009  9:55:30.051711
```

OS system time can also be retrieved by `system_time/0` and `system_time/1`.

```
perf_counter() -> Counter
```

Types:
```
Counter = integer()
```

Returns the current performance counter value in `perf_counter` time unit. This is a highly optimized call that might not be traceable.

```
perf_counter(Unit) -> integer()
```

Types:
```
Unit = erlang:time_unit()
```

Returns a performance counter that can be used as a very fast and high resolution timestamp. This counter is read directly from the hardware or operating system with the same guarantees. This means that two consecutive calls to the function are not guaranteed to be monotonic, though it most likely will be. The performance counter will be converted to the resolution passed as an argument.

```
1> T1 = os:perf_counter(1000),receive after 10000 -> ok end,T2 = os:perf_counter(1000).
176525861
2> T2 - T1.
10004
```

```
type() -> {Osfamily, Osname}
```

Types:
```
Osfamily = unix | win32
Osnmae = atom()
```

Returns the `Osfamily` and, in some cases, the `Osname` of the current OS.

On Unix, `Osname` has the same value as `uname -s` returns, but in lower case. For example, on Solaris 1 and 2, it is `sunos`.

On Windows, `Osname` is `nt`.

**Note:**

Think twice before using this function. Use module `filename` if you want to inspect or build filenames in a portable way. Avoid matching on atom `Osname`.
unsetenv(VarName) -> true
Types:
    VarName = env_var_name()
Deletes the environment variable VarName.
If Unicode filename encoding is in effect (see the \texttt{erl} manual page), the string \texttt{VarName} can contain characters with codepoints \textgreater{} 255.

\textbf{version()} \rightarrow \text{VersionString | \{Major, Minor, Release\}}
Types:
    VersionString = string()
    Major = Minor = Release = integer() \geq 0
Returns the OS version. On most systems, this function returns a tuple, but a string is returned instead if the system has versions that cannot be expressed as three numbers.

\textbf{Note:}
Think twice before using this function. If you still need to use it, always call \texttt{os:version()} first.
Erlang module

This module implements process groups. A message can be sent to one, some, or all group members.

Up until OTP 17 there used to exist an experimental pg module in stdlib. This pg module is not the same module as that experimental pg module, and only share the same module name.

A group of processes can be accessed by a common name. For example, if there is a group named foobar, there can be a set of processes (which can be located on different nodes) that are all members of the group foobar. There are no special functions for sending a message to the group. Instead, client functions are to be written with the functions get_members/1 and get_local_members/1 to determine which processes are members of the group. Then the message can be sent to one or more group members.

If a member terminates, it is automatically removed from the group.

A process may join multiple groups. It may join the same group multiple times. It is only allowed to join processes running on local node.

Process Groups implement strong eventual consistency. Unlike pg2, that provides strong ordering guarantees, Process Groups membership view may temporarily diverge. For example, when processes on node1 and node2 join concurrently, node3 and node4 may receive updates in a different order.

Membership view is not transitive. If node1 is not directly connected to node2, they will not see each other groups. But if both are connected to node3, node3 will have the full view.

Groups are automatically created when any process joins, and are removed when all processes leave the group. Non-existing group is considered empty (containing no processes).

Process groups can be organised into multiple scopes. Scopes are completely independent of each other. A process may join any number of groups in any number of scopes. Scopes are designed to decouple single mesh into a set of overlay networks, reducing amount of traffic required to propagate group membership information. Default scope pg is started automatically when kernel(6) is configured to do so.

### Note:

Scope name is used to register process locally, and to name an ETS table. If there is another process registered under this name, or another ETS table exists, scope fails to start.

Local membership is not preserved if scope process exits and restarts. This behaviour is different from pg2, that recovers local membership from remote nodes.

### Data Types

```erlang
group() = any()
```

The identifier of a process group.

### Exports

```erlang
start_link() -> {ok, pid()} | {error, any()}
```

Starts the default pg scope within supervision tree. Kernel may be configured to do it automatically, see kernel(6) configuration manual.
start(Scope :: atom()) -> {ok, pid()} | {error, any()}
start_link(Scope :: atom()) -> {ok, pid()} | {error, any()}

Starts additional scope.

join(Group :: group(), PidOrPids :: pid() | [pid()]) -> ok
join(Scope :: atom(),
    Group :: group(),
    PidOrPids :: pid() | [pid()]) ->
    ok

Joins single process or multiple processes to the group Name. A process can join a group many times and must then leave the group the same number of times.

PidOrPids may contain the same process multiple times.

leave(Group :: group(), PidOrPids :: pid() | [pid()]) -> ok
leave(Scope :: atom(),
    Group :: group(),
    PidOrPids :: pid() | [pid()]) ->
    ok | notJoined

Makes the process PidOrPids leave the group Name. If the process is not a member of the group, notJoined is returned.

When list of processes is passed as PidOrPids function returns notJoined only when all processes of the list are not joined.

get_local_members(Group :: group()) -> [pid()]
get_local_members(Scope :: atom(), Group :: group()) -> [pid()]

Returns all processes running on the local node in the group Name. Processes are returned in no specific order. This function is optimised for speed.

get_members(Group :: group()) -> [pid()]
get_members(Scope :: atom(), Group :: group()) -> [pid()]

Returns all processes in the group Name. Processes are returned in no specific order. This function is optimised for speed.

which_groups() -> [Group :: group()]
which_groups(Scope :: atom()) -> [Group :: group()]

Returns a list of all known groups.

See Also

kernel(6)
Erlang module

**Warning:**
The pg2 module is deprecated as of OTP 23 and scheduled for removal in OTP 24. You are advised to replace the usage of pg2 with pg. pg has a similar API, but with an implementation that is more scalable. See the documentation of pg for more information about differences.

This module implements process groups. Each message can be sent to one, some, or all group members.

A group of processes can be accessed by a common name. For example, if there is a group named foobar, there can be a set of processes (which can be located on different nodes) that are all members of the group foobar. There are no special functions for sending a message to the group. Instead, client functions are to be written with the functions get_members/1 and get_local_members/1 to determine which processes are members of the group. Then the message can be sent to one or more group members.

If a member terminates, it is automatically removed from the group.

**Warning:**
This module is used by module disk_log for managing distributed disk logs. The disk log names are used as group names, which means that some action can be needed to avoid name clashes.

**Data Types**

name() = any()  
The name of a process group.

**Exports**

create(Name :: name()) -> ok  
Creates a new, empty process group. The group is globally visible on all nodes. If the group exists, nothing happens.

delete(Name :: name()) -> ok  
Deletes a process group.

get_closest_pid(Name) -> pid() | {error, Reason}  
Types:
Name = name()  
Reason = {no_process, Name} | {no_such_group, Name}  
A useful dispatch function that can be used from client functions. It returns a process on the local node, if such a process exists. Otherwise, it selects one randomly.

get_local_members(Name) ->  
[pid()] | {error, {no_such_group, Name}}  
Types:
Name = name()

Returns all processes running on the local node in the group Name. This function is to be used from within a client function that accesses the group. It is therefore optimized for speed.

get_members(Name) -> [pid()] | {error, {no_such_group, Name}}

Types:
  Name = name()

Returns all processes in the group Name. This function is to be used from within a client function that accesses the group. It is therefore optimized for speed.

join(Name, Pid :: pid()) -> ok | {error, {no_such_group, Name}}

Types:
  Name = name()

Joins the process Pid to the group Name. A process can join a group many times and must then leave the group the same number of times.

leave(Name, Pid :: pid()) -> ok | {error, {no_such_group, Name}}

Types:
  Name = name()

Makes the process Pid leave the group Name. If the process is not a member of the group, ok is returned.

start() -> {ok, pid()} | {error, any()}
start_link() -> {ok, pid()} | {error, any()}

Starts the pg2 server. Normally, the server does not need to be started explicitly, as it is started dynamically if it is needed. This is useful during development, but in a target system the server is to be started explicitly. Use the configuration parameters for kernel(6) for this.

which_groups() -> [Name :: name()]

Returns a list of all known groups.

See Also

kernel(6)
This module contains services similar to Remote Procedure Calls. It also contains broadcast facilities and parallel evaluators. A remote procedure call is a method to call a function on a remote node and collect the answer. It is used for collecting information on a remote node, or for running a function with some specific side effects on the remote node.

**Note:**

`rpc:call()` and friends makes it quite hard to distinguish between successful results, raised exceptions, and other errors. This cannot be changed due to compatibility reasons. As of OTP 23, a new module `erpc` was introduced in order to provide an API that makes it possible to distinguish between the different results. The `erpc` module provides a subset (however, the central subset) of the functionality available in the `rpc` module. The `erpc` implementation also provides a more scalable implementation with better performance than the original `rpc` implementation. However, since the introduction of `erpc`, the `rpc` module implements large parts of its central functionality using `erpc`, so the `rpc` module wont not suffer scalability wise and performance wise compared to `erpc`.

**Data Types**

**key()**

Opaque value returned by `async_call/4`.

**Exports**

**abcast(Name, Msg) -> abcast**

*Types:*

- `Name = atom()`
- `Msg = term()`

Equivalent to `abcast([node()|nodes()], Name, Msg)`.

**abcast(Nodes, Name, Msg) -> abcast**

*Types:*

- `Nodes = [node()]`
- `Name = atom()`
- `Msg = term()`

Broadcasts the message `Msg` asynchronously to the registered process `Name` on the specified nodes.

**async_call(Node, Module, Function, Args) -> Key**

*Types:*

---
Node = node()
Module = module()
Function = atom()
Args = [term()]
Key = key()

Implements **call streams with promises**, a type of RPC that does not suspend the caller until the result is finished. Instead, a key is returned, which can be used later to collect the value. The key can be viewed as a promise to deliver the answer.

In this case, the key `Key` is returned, which can be used in a subsequent call to `yield/1` or `nb_yield/1, 2` to retrieve the value of evaluating `apply(Module, Function, Args)` on node `Node`.

**Note:**

If you want the ability to distinguish between results, you may want to consider using the `erpc:send_request()` function from the `erpc` module instead. This also gives you the ability retrieve the results in other useful ways.

**Note:**

`yield/1` and `nb_yield/1, 2` must be called by the same process from which this function was made otherwise they will never yield correctly.

**Note:**

You cannot make any assumptions about the process that will perform the `apply()`. It may be an `rpc` server, another server, or a freshly spawned process.

\[
\text{block_call(Node, Module, Function, Args)} \rightarrow \text{Res | {badrpc, Reason}}
\]

Types:

- Node = node()
- Module = module()
- Function = atom()
- Args = [term()]
- Res = Reason = term()

The same as calling `rpc:block_call(Node, Module, Function, Args, infinity)`.

\[
\text{block_call(Node, Module, Function, Args, Timeout)} \rightarrow \text{Res | {badrpc, Reason}}
\]

Types:
Node = node()
Module = module()
Function = atom()
Args = [term()]
Res = Reason = term()
Timeout = 0..4294967295 | infinity

The same as calling rpc:call(Node, Module, Function, Args, Timeout) with the exception that it also blocks other rpc:block_call() operations from executing concurrently on the node Node.

Warning:

Note that it also blocks other operations than just rpc:block_call() operations, so use it with care.

call(Node, Module, Function, Args) -> Res | {badrpc, Reason}

Types:

Node = node()
Module = module()
Function = atom()
Args = [term()]
Res = Reason = term()

Evaluates apply(Module, Function, Args) on node Node and returns the corresponding value Res, or {badrpc, Reason} if the call fails. The same as calling rpc:call(Node, Module, Function, Args, infinity).

call(Node, Module, Function, Args, Timeout) ->
    Res | {badrpc, Reason}

Types:

Node = node()
Module = module()
Function = atom()
Args = [term()]
Res = Reason = term()
Timeout = 0..4294967295 | infinity

Evaluates apply(Module, Function, Args) on node Node and returns the corresponding value Res, or {badrpc, Reason} if the call fails. Timeout is a time-out value in milliseconds. If the call times out, Reason is timeout.

If the reply arrives after the call times out, no message contaminates the caller's message queue.

Note:

If you want the ability to distinguish between results, you may want to consider using the erpc:call() function from the erpc module instead.
Note:
Here follows the details of what exactly is returned.
(badrpc, Reason) will be returned in the following circumstances:
• The called function fails with an exit exception.
• The called function fails with an error exception.
• The called function returns a term that matches {'EXIT', _}.
• The called function throws a term that matches {'EXIT', _}.
Res is returned in the following circumstances:
• The called function returns normally with a term that does not match {'EXIT', _}.
• The called function throws a term that does not match {'EXIT', _}.

Note:
You cannot make any assumptions about the process that will perform the apply(). It may be the calling process itself, an rpc server, another server, or a freshly spawned process.

cast(Node, Module, Function, Args) -> true
Types:
Node = node()
Module = module()
Function = atom()
Args = [term()]
Evaluates apply(Module, Function, Args) on node Node. No response is delivered and the calling process is not suspended until the evaluation is complete, as is the case with call/4, 5.

Note:
You cannot make any assumptions about the process that will perform the apply(). It may be an rpc server, another server, or a freshly spawned process.

eval_everywhere(Module, Function, Args) -> abcast
Types:
Module = module()
Function = atom()
Args = [term()]
Equivalent to eval_everywhere([node() | nodes()], Module, Function, Args).

eval_everywhere(Nodes, Module, Function, Args) -> abcast
Types:
rpc

    Nodes = [node()]
    Module = module()
    Function = atom()
    Args = [term()]
Evaluates apply(Module, Function, Args) on the specified nodes. No answers are collected.

multi_server_call(Name, Msg) -> {Replies, BadNodes}
Types:
    Name = atom()
    Msg = term()
    Replies = [Reply :: term()]
    BadNodes = [node()]
Equivalent to multi_server_call([node()|nodes()], Name, Msg).

multi_server_call(Nodes, Name, Msg) -> {Replies, BadNodes}
Types:
    Nodes = [node()]
    Name = atom()
    Msg = term()
    Replies = [Reply :: term()]
    BadNodes = [node()]
Can be used when interacting with servers called Name on the specified nodes. It is assumed that the servers receive messages in the format {From, Msg} and reply using From ! {Name, Node, Reply}, where Node is the name of the node where the server is located. The function returns {Replies, BadNodes}, where Replies is a list of all Reply values, and BadNodes is one of the following:

- A list of the nodes that do not exist
- A list of the nodes where the server does not exist
- A list of the nodes where the server terminated before sending any reply.

multicall(Module, Function, Args) -> {ResL, BadNodes}
Types:
    Module = module()
    Function = atom()
    Args = [term()]
    ResL = [Res :: term() | {badrpc, Reason :: term()}]
    BadNodes = [node()]
Equivalent to multicall([node()|nodes()], Module, Function, Args, infinity).

multicall(Nodes, Module, Function, Args) -> {ResL, BadNodes}
Types:
Nodes = [node()]
Module = module()
Function = atom()
Args = [term()]
ResL = [Res :: term() | {badrpc, Reason :: term()}]
BadNodes = [node()]

Equivalent to multicall(Nodes, Module, Function, Args, infinity).

\[\text{multicall}(Module, Function, Args, Timeout) \rightarrow \{\text{ResL}, \text{BadNodes}\}\]

Types:
\[
\begin{align*}
\text{Module} &= \text{module()} \\
\text{Function} &= \text{atom()} \\
\text{Args} &= [\text{term()}] \\
\text{Timeout} &= 0..4294967295 | \text{infinity} \\
\text{ResL} &= [\text{Res} :: \text{term()} | \{\text{badrpc}, \text{Reason} :: \text{term()}\}] \\
\text{BadNodes} &= [\text{node()}]
\end{align*}
\]

Equivalent to multicall([node()]|nodes()), Module, Function, Args, Timeout).

\[\text{multicall}(\text{Nodes}, \text{Module}, \text{Function}, \text{Args}, \text{Timeout}) \rightarrow \{\text{ResL}, \text{BadNodes}\}\]

Types:
\[
\begin{align*}
\text{Nodes} &= [\text{node()}] \\
\text{Module} &= \text{module()} \\
\text{Function} &= \text{atom()} \\
\text{Args} &= [\text{term()}] \\
\text{Timeout} &= 0..4294967295 | \text{infinity} \\
\text{ResL} &= [\text{Res} :: \text{term()} | \{\text{badrpc}, \text{Reason} :: \text{term()}\}] \\
\text{BadNodes} &= [\text{node()}]
\end{align*}
\]

In contrast to an RPC, a multicall is an RPC that is sent concurrently from one client to multiple servers. This is useful for collecting information from a set of nodes, or for calling a function on a set of nodes to achieve some side effects. It is semantically the same as iteratively making a series of RPCs on all the nodes, but the multicall is faster, as all the requests are sent at the same time and are collected one by one as they come back.

The function evaluates apply(Module, Function, Args) on the specified nodes and collects the answers. It returns \{ResL, BadNodes\}, where BadNodes is a list of the nodes that do not exist, and ResL is a list of the return values, or \{badrpc, Reason\} for failing calls. Timeout is a time (integer) in milliseconds, or infinity.

The following example is useful when new object code is to be loaded on all nodes in the network, and indicates some side effects that RPCs can produce:

```erlang
%% Find object code for module Mod
{Mod, Bin, File} = code:get_object_code(Mod),

%% and load it on all nodes including this one
{ResL, _} = rpc:multicall(code, load_binary, [Mod, File, Bin]),

%% and then maybe check the ResL list.
```
Note:
If you want the ability to distinguish between results, you may want to consider using the `erpc:multicall()` function from the `erpc` module instead.

Note:
You cannot make any assumptions about the process that will perform the `apply()`. It may be the calling process itself, an `rpc` server, another server, or a freshly spawned process.

```erlang
cpy_nb_yield(Key) -> {value, Val} | timeout

Types:
  Key = key()
  Val = (Res :: term()) | {badrpc, Reason :: term()}

Equivalent to `nb_yield(Key, 0)`.

ncpy_nb_yield(Key, Timeout) -> {value, Val} | timeout

Types:
  Key = key()
  Timeout = 0..4294967295 | infinity
  Val = (Res :: term()) | {badrpc, Reason :: term()}

Non-blocking version of `yield/1`. It returns the tuple `{value, Val}` when the computation is finished, or `timeout` when `Timeout` milliseconds has elapsed.

See the note in `call/4` for more details of `Val`.

Note:
This function must be called by the same process from which `async_call/4` was made otherwise it will only return `timeout`.

parallel_eval(FuncCalls) -> ResL

Types:
  FuncCalls = [{Module, Function, Args}]
              Module = module()
              Function = atom()
              Args = ResL = [term()]

Evaluates, for every tuple in `FuncCalls, apply(Module, Function, Args)` on some node in the network. Returns the list of return values, in the same order as in `FuncCalls`.

```
Pid = pid()
Item = atom()
Info = term()

Location transparent version of the BIF `erlang:process_info/1` in ERTS.

\[ \text{pinfo}(\text{Pid}, \text{Item}) \rightarrow \{\text{Item, Info}\} \mid \text{undefined} \mid [] \]
\[ \text{pinfo}(\text{Pid}, \text{ItemList}) \rightarrow \{[\text{Item, Info}]\} \mid \text{undefined} \mid [] \]

Types:
- Pid = pid()
- Item = atom()
- ItemList = [Item]
- Info = term()

Location transparent version of the BIF `erlang:process_info/2` in ERTS.

\[ \text{pmap}(\text{FuncSpec}, \text{ExtraArgs}, \text{List1}) \rightarrow \text{List2} \]

Types:
- FuncSpec = \{\text{Module, Function}\}
- Module = module()
- Function = atom()
- ExtraArgs = [\text{term()}]
- List1 = [\text{Elem} :: \text{term()}]
- List2 = [\text{term()}]

Evaluates `apply(Module, Function, [Elem|ExtraArgs])` for every element `Elem` in `List1`, in parallel. Returns the list of return values, in the same order as in `List1`.

\[ \text{sbcast}(\text{Name}, \text{Msg}) \rightarrow \{\text{GoodNodes, BadNodes}\} \]

Types:
- Name = atom()
- Msg = term()
- GoodNodes = BadNodes = [node()]

Equivalent to `sbcast([\text{node()}|\text{nodes()}], \text{Name}, \text{Msg})`.

\[ \text{sbcast}(\text{Nodes, Name, Msg}) \rightarrow \{\text{GoodNodes, BadNodes}\} \]

Types:
- Name = atom()
- Msg = term()
- Nodes = GoodNodes = BadNodes = [node()]

Broadcasts the message `Msg` synchronously to the registered process `Name` on the specified nodes. Returns `{GoodNodes, BadNodes}`, where `GoodNodes` is the list of nodes that have `Name` as a registered process. The function is synchronous in the sense that it is known that all servers have received the message when the call returns. It is not possible to know that the servers have processed the message.

Any further messages sent to the servers, after this function has returned, are received by all servers after this message.
server_call(Node, Name, ReplyWrapper, Msg) ->
    Reply | {error, Reason}
Types:
    Node = node()
    Name = atom()
    ReplyWrapper = Msg = Reply = term()
    Reason = nodedown
Can be used when interacting with a server called Name on node Node. It is assumed that the server receives messages in the format {From, Msg} and replies using From ! {ReplyWrapper, Node, Reply}. This function makes such a server call and ensures that the entire call is packed into an atomic transaction, which either succeeds or fails. It never hangs, unless the server itself hangs.

The function returns the answer Reply as produced by the server Name, or {error, Reason}.

yield(Key) -> Res | {badrpc, Reason}
Types:
    Key = key()
    Res = Reason = term()
Returns the promised answer from a previous async_call/4. If the answer is available, it is returned immediately. Otherwise, the calling process is suspended until the answer arrives from Node.

Note:
This function must be called by the same process from which async_call/4 was made otherwise it will never return.

See the note in call/4 for more details of the return value.
seq_trace

Erlang module

Sequential tracing makes it possible to trace information flows between processes resulting from one initial transfer of information. Sequential tracing is independent of the ordinary tracing in Erlang, which is controlled by the \texttt{erlang:trace/3} BIF. For more information about what sequential tracing is and how it can be used, see section Sequential Tracing.

\texttt{seq_trace} provides functions that control all aspects of sequential tracing. There are functions for activation, deactivation, inspection, and for collection of the trace output.

Data Types

\texttt{token()} = \{integer(), boolean(), term(), term(), term()\}

An opaque term (a tuple) representing a trace token.

Exports

\texttt{set_token(Token) -> PreviousToken | ok}

Types:

\texttt{Token = PreviousToken} = \[] | \texttt{token()}

Sets the trace token for the calling process to \texttt{Token}. If \texttt{Token = \[]} then tracing is disabled, otherwise \texttt{Token} should be an Erlang term returned from \texttt{get_token/0} or \texttt{set_token/1}. \texttt{set_token/1} can be used to temporarily exclude message passing from the trace by setting the trace token to empty like this:

```erlang
OldToken = seq_trace:set_token([], % set to empty and save
  % old value
  io:format("Exclude the signalling caused by this-\n")
  seq_trace:set_token(OldToken), % activate the trace token again
...
```

Returns the previous value of the trace token.

\texttt{set_token(Component, Val) -> OldVal}

Types:

\texttt{Component = component()}
\texttt{Val = OldVal} = \texttt{value()}
\texttt{component()} = \texttt{label} | \texttt{serial} | \texttt{flag()}
\texttt{flag()} =
  \texttt{send} | \texttt{'receive'} | \texttt{print} | \texttt{timestamp} | \texttt{monotonic_timestamp} | \texttt{strict_monotonic_timestamp}
\texttt{value()} =
  \{(Label :: term()) \ |
  \{Previous :: integer() >= 0, Current :: integer() >= 0} \ |
  \{Bool :: boolean()\}

Sets the individual \texttt{Component} of the trace token to \texttt{Val}. Returns the previous value of the component.
**seq_trace**

**set_token(label, Label)**

The `label` component is a term which identifies all events belonging to the same sequential trace. If several sequential traces can be active simultaneously, `label` is used to identify the separate traces. Default is 0.

**Warning:**

Labels were restricted to small signed integers (28 bits) prior to OTP 21. The trace token will be silently dropped if it crosses over to a node that does not support the label.

**set_token(serial, SerialValue)**

`SerialValue = {Previous, Current}`. The `serial` component contains counters which enables the traced messages to be sorted, should never be set explicitly by the user as these counters are updated automatically. Default is `{0, 0}`.

**set_token(send, Bool)**

A trace token flag (`true | false`) which enables/disables tracing on information sending. Default is `false`.

**set_token('receive', Bool)**

A trace token flag (`true | false`) which enables/disables tracing on information reception. Default is `false`.

**set_token(print, Bool)**

A trace token flag (`true | false`) which enables/disables tracing on explicit calls to `seq_trace:print/1`. Default is `false`.

**set_token(timestamp, Bool)**

A trace token flag (`true | false`) which enables/disables a timestamp to be generated for each traced event. Default is `false`.

**set_token(strict_monotonic_timestamp, Bool)**

A trace token flag (`true | false`) which enables/disables a strict monotonic timestamp to be generated for each traced event. Default is `false`. Timestamps will consist of Erlang monotonic time and a monotonically increasing integer. The time-stamp has the same format and value as produced by `{erlang:monotonic_time(nanosecond), erlang:unique_integer([monotonic])}`.

**set_token(monotonic_timestamp, Bool)**

A trace token flag (`true | false`) which enables/disables a strict monotonic timestamp to be generated for each traced event. Default is `false`. Timestamps will use Erlang monotonic time. The time-stamp has the same format and value as produced by `erlang:monotonic_time(nanosecond)`.

If multiple timestamp flags are passed, `timestamp` has precedence over `strict_monotonic_timestamp` which in turn has precedence over `monotonic_timestamp`. All timestamp flags are remembered, so if two are passed and the one with highest precedence later is disabled the other one will become active.

**get_token() -> [] | token()**

Returns the value of the trace token for the calling process. If `[]` is returned, it means that tracing is not active. Any other value returned is the value of an active trace token. The value returned can be used as input to the `set_token/1` function.

**get_token(Component) -> {Component, Val}**

Types:
Component = component()
Val = value()
component() = label | serial | flag()
flag() = send | 'receive' | print | timestamp | monotonic_timestamp | strict_monotonic_timestamp
value() =
(Label :: term()) |
{Previous :: integer() >= 0, Current :: integer() >= 0} |
{Bool :: boolean()}

Returns the value of the trace token component Component. See set_token/2 for possible values of Component and Val.

print(TraceInfo) -> ok
Types:
   TraceInfo = term()

Puts the Erlang term TraceInfo into the sequential trace output if the calling process currently is executing within a sequential trace and the print flag of the trace token is set.

print(Label, TraceInfo) -> ok
Types:
   Label = integer()
   TraceInfo = term()

Same as print/1 with the additional condition that TraceInfo is output only if Label is equal to the label component of the trace token.

reset_trace() -> true

Sets the trace token to empty for all processes on the local node. The process internal counters used to create the serial of the trace token is set to 0. The trace token is set to empty for all messages in message queues. Together this will effectively stop all ongoing sequential tracing in the local node.

set_system_tracer(Tracer) -> OldTracer
Types:
   Tracer = OldTracer = tracer()
   tracer() = 
   (Pid :: pid()) |
   port() |
   (TracerModule :: {module(), term()}) |
   false

Sets the system tracer. The system tracer can be either a process, port or tracer module denoted by Tracer. Returns the previous value (which can be false if no system tracer is active).
Failure: {badarg, Info} if Pid is not an existing local pid.

get_system_tracer() -> Tracer
Types:
Tracer = tracer()
tracer() =
  (Pid :: pid()) |
  port() |
  (TracerModule :: {module(), term()}) |
  false

Returns the pid, port identifier or tracer module of the current system tracer or false if no system tracer is activated.

Trace Messages Sent to the System Tracer

The format of the messages is one of the following, depending on if flag timestamp of the trace token is set to true or false:

{seq_trace, Label, SeqTraceInfo, TimeStamp}

or

{seq_trace, Label, SeqTraceInfo}

Where:

Label = int()
TimeStamp = {Seconds, Milliseconds, Microseconds}
Seconds = Milliseconds = Microseconds = int()

SeqTraceInfo can have the following formats:

{send, Serial, From, To, Message}

Used when a process From with its trace token flag send set to true has sent information. To may be a process identifier, a registered name on a node represented as {NameAtom, NodeAtom}, or a node name represented as an atom. From may be a process identifier or a node name represented as an atom. Message contains the information passed along in this information transfer. If the transfer is done via message passing, it is the actual message.

{'receive', Serial, From, To, Message}

Used when a process To receives information with a trace token that has flag 'receive' set to true. To may be a process identifier or a node name represented as an atom. From may be a process identifier or a node name represented as an atom. Message contains the information passed along in this information transfer. If the transfer is done via message passing, it is the actual message.

{print, Serial, From, _, Info}

Used when a process From has called seq_trace:print(Label, TraceInfo) and has a trace token with flag print set to true, and label set to Label.

Serial is a tuple (PreviousSerial, ThisSerial), where:

• Integer PreviousSerial denotes the serial counter passed in the last received information that carried a trace token. If the process is the first in a new sequential trace, PreviousSerial is set to the value of the process internal "trace clock".

• Integer ThisSerial is the serial counter that a process sets on outgoing messages. It is based on the process internal "trace clock", which is incremented by one before it is attached to the trace token in the message.
Sequential Tracing

Sequential tracing is a way to trace a sequence of information transfers between different local or remote processes, where the sequence is initiated by a single transfer. The typical information transfer is an ordinary Erlang message passed between two processes, but information is transferred also in other ways. In short, it works as follows:

Each process has a trace token, which can be empty or not empty. When not empty, the trace token can be seen as the tuple \( \{ \text{Label}, \text{Flags}, \text{Serial}, \text{From} \} \). The trace token is passed invisibly when information is passed between processes. In most cases the information is passed in ordinary messages between processes, but information is also passed between processes by other means. For example, by spawning a new process. An information transfer between two processes is represented by a send event and a receive event regardless of how it is passed.

To start a sequential trace, the user must explicitly set the trace token in the process that will send the first information in a sequence.

The trace token of a process is set each time the process receives information. This is typically when the process matches a message in a receive statement, according to the trace token carried by the received message, empty or not.

On each Erlang node, a process can be set as the **system tracer**. This process will receive trace messages each time information with a trace token is sent or received (if the trace token flag `send` or `'receive'` is set). The system tracer can then print each trace event, write it to a file, or whatever suitable.

**Note:**

The system tracer only receives those trace events that occur locally within the Erlang node. To get the whole picture of a sequential trace, involving processes on many Erlang nodes, the output from the system tracer on each involved node must be merged (offline).

The following sections describe sequential tracing and its most fundamental concepts.

Different Information Transfers

Information flows between processes in a lot of different ways. Not all flows of information will be covered by sequential tracing. One example is information passed via ETS tables. Below is a list of information paths that are covered by sequential tracing:

**Message Passing**

All ordinary messages passed between Erlang processes.

**Exit signals**

An exit signal is represented as an \( \{'EXIT', \text{Pid}, \text{Reason} \} \) tuple.

**Process Spawn**

A process spawn is represented as multiple information transfers. At least one spawn request and one spawn reply. The actual amount of information transfers depends on what type of spawn it is and may also change in future implementations. Note that this is more or less an internal protocol that you are peeking at. The spawn request will be represented as a tuple with the first element containing the atom `spawn_request`, but this is more or less all that you can depend on.

**Note:**

If you do ordinary `send` or `receive` trace on the system, you will only see ordinary message passing, not the other information transfers listed above.
Trace Token

Each process has a current trace token which is "invisibly" passed from the parent process on creation of the process. The current token of a process is set in one of the following two ways:

• Explicitly by the process itself, through a call to seq_trace:set_token/1,2
• When information is received. This is typically when a received message is matched out in a receive expression, but also when information is received in other ways.

In both cases, the current token is set. In particular, if the token of a received message is empty, the current token of the process is set to empty.

A trace token contains a label and a set of flags. Both the label and the flags are set in both alternatives above.

Serial

The trace token contains a component called serial. It consists of two integers, Previous and Current. The purpose is to uniquely identify each traced event within a trace sequence, as well as to order the messages chronologically and in the different branches, if any.

The algorithm for updating Serial can be described as follows:

Let each process have two counters, prev_cnt and curr_cnt, both are set to 0 when a process is created outside of a trace sequence. The counters are updated at the following occasions:

• When the process is about to pass along information to another process and the trace token is not empty. This typically occurs when sending a message, but also, for example, when spawning another process.

Let the serial of the trace token be tprev and tcurr.

```
curr_cnt := curr_cnt + 1
tprev := prev_cnt
tcurr := curr_cnt
```

The trace token with tprev and tcurr is then passed along with the information passed to the other process.

• When the process calls seq_trace:print(Label, Info). Label matches the label part of the trace token and the trace token print flag is true.

The algorithm is the same as for send above.

• When information is received that also contains a non-empty trace token. For example, when a message is matched out in a receive expression, or when a new process is spawned.

The process trace token is set to the trace token from the message.

Let the serial of the trace token be tprev and tcurr.

```
if (curr_cnt < tcurr )
curr_cnt := tcurr
prev_cnt := tcurr
```

curr_cnt of a process is incremented each time the process is involved in a sequential trace. The counter can reach its limit (27 bits) if a process is very long-lived and is involved in much sequential tracing. If the counter overflows,
the serial for ordering of the trace events cannot be used. To prevent the counter from overflowing in the middle of a sequential trace, function seq_trace:reset_trace/0 can be called to reset prev_cnt and curr_cnt of all processes in the Erlang node. This function also sets all trace tokens in processes and their message queues to empty, and thus stops all ongoing sequential tracing.

Performance Considerations

The performance degradation for a system that is enabled for sequential tracing is negligible as long as no tracing is activated. When tracing is activated, there is an extra cost for each traced message, but all other messages are unaffected.

Ports

Sequential tracing is not performed across ports.

If the user for some reason wants to pass the trace token to a port, this must be done manually in the code of the port controlling process. The port controlling processes have to check the appropriate sequential trace settings (as obtained from seq_trace:get_token/1) and include trace information in the message data sent to their respective ports.

Similarly, for messages received from a port, a port controller has to retrieve trace-specific information, and set appropriate sequential trace flags through calls to seq_trace:set_token/2.

Distribution

Sequential tracing between nodes is performed transparently. This applies to C-nodes built with Erl_Interface too. A C-node built with Erl_Interface only maintains one trace token, which means that the C-node appears as one process from the sequential tracing point of view.

Example of Use

This example gives a rough idea of how the new primitives can be used and what kind of output it produces.

Assume that you have an initiating process with Pid == <0.30.0> like this:

```
-module(seqex).
-compile(export_all).

loop(Port) ->
    receive
        {Port,Message} ->
            seq_trace:set_token(label,17),
            seq_trace:set_token('receive',true),
            seq_trace:set_token(print,true),
            seq_trace:print(17,"**** Trace Started *****"),
            call_server ! {self(),the_message);
        {ack,Ack} ->
            ok
    end,
    loop(Port).
```

And a registered process call_server with Pid == <0.31.0> like this:

```
loop() ->
    receive
        {PortController,Message} ->
            Ack = {received, Message},
            seq_trace:print(17,"We are here now"),
            PortController ! {ack,Ack}
    end,
    loop().
```
A possible output from the system's `sequential_tracer` can be like this:

```
17:<0.30.0> Info {0,1} WITH
"**** Trace Started ****"
17:<0.31.0> Received {0,2} FROM <0.30.0> WITH
{<0.30.0>,the_message}
17:<0.31.0> Info {2,3} WITH
"We are here now"
17:<0.30.0> Received {2,4} FROM <0.31.0> WITH
{ack,{received,the_message}}
```

The implementation of a system tracer process that produces this printout can look like this:

```
tracer() ->
  receive
    {seq_trace,Label,TraceInfo} ->
      print_trace(Label,TraceInfo,false);
    {seq_trace,Label,TraceInfo,Ts} ->
      print_trace(Label,TraceInfo,Ts);
    _Other -> ignore
  end,
  tracer().

print_trace(Label,TraceInfo,false) ->
  io:format("~p:",[Label]),
  print_trace(TraceInfo);
print_trace(Label,TraceInfo,Ts) ->
  io:format("~p ~p:",[Label,Ts]),
  print_trace(TraceInfo).

print_trace({print,Serial,From,_,Info}) ->
  io:format("~p Info ~p WITH~n~p~n",[From,Serial,Info]);
print_trace({'receive',Serial,From,To,Message}) ->
  io:format("~p Received ~p FROM ~p WITH~n~p~n",[To,Serial,From,Message]);
print_trace({send,Serial,From,To,Message}) ->
  io:format("~p Sent ~p TO ~p WITH~n~p~n",[From,Serial,To,Message]).
```

The code that creates a process that runs this tracer function and sets that process as the system tracer can look like this:

```
start() ->
  Pid = spawn(?MODULE,tracer,[]),
  seq_trace:set_system_tracer(Pid), % set Pid as the system tracer
  ok.
```

With a function like `test/0`, the whole example can be started:

```
test() ->
  P = spawn(?MODULE, loop, [port]),
  register(call_server, spawn(?MODULE, loop, [])),
  start(),
  P ! {port,message}.
```
This module provides an API for network socket. Functions are provided to create, delete and manipulate the sockets as well as sending and receiving data on them.

The intent is that it shall be as "close as possible" to the OS level socket interface. The only significant addition is that some of the functions, e.g. `recv/3`, has a timeout argument.

### Note:

Some functions allow for an asynchronous call. This is achieved by setting the Timeout argument to `nowait`. For instance, if calling the `recv/3` function with Timeout set to `nowait` (`recv(Sock, 0, nowait)`) when there is actually nothing to read, it will return with `{select, SelectInfo}` (`.SelectInfo` contains the `SelectRef`). When data eventually arrives a 'select' message will be sent to the caller:

```erlang
{_sock, socket(), select, SelectRef}
```

The caller can now make another call to the `recv` function and now expect data.

Note that all other users are locked out until the 'current user' has called the function (recv in this case).

Another message the user must be prepared for (when making asynchronous calls) is the abort message:

```erlang
{_sock, socket(), abort, Info}
```

This message indicates that the (asynchronous) operation has been aborted. If, for instance, the socket has been closed (by another process). `Info` will be `{SelectRef, closed}`.

### Note:

There is currently no support for Windows.

Support for IPv6 has been implemented but not tested.

SCTP has only been partly implemented (and not tested).

### Data Types

- `domain() = local | inet | inet6`
- `type() = stream | dgram | raw | rdm | seqpacket`
- `protocol() = ip | tcp | udp | sctp | icmp | igmp | {raw, integer()}`
- `socket() = {socket, domain(), type(), protocol()}`
- `As returned by open/1,2,3,4 and accept/1,2`.
- `select_tag() = {select_tag, select_ref()}`
- `select_ref() = {select_ref, select_tag()}`
- `select_info() = {select_info, select_tag(), select_ref()}`
- `socket_counters() =`
socket

#{read_byte := integer() >= 0,
  read_fails := integer() >= 0,
  read_pkg := integer() >= 0,
  read_pkg_max := integer() >= 0,
  read_tries := integer() >= 0,
  read_waits := integer() >= 0,
  write_byte := integer() >= 0,
  write_fails := integer() >= 0,
  write_pkg := integer() >= 0,
  write_pkg_max := integer() >= 0,
  write_tries := integer() >= 0,
  write_waits := integer() >= 0,
  acc_success := integer() >= 0,
  acc_fails := integer() >= 0,
  acc_tries := integer() >= 0,
  acc_waits := integer() >= 0}

socket_info() =
#{domain := domain(),
  type := type(),
  protocol := protocol(),
  ctrl := pid(),
  ctype := normal | fromfd | {fromfd, integer()},
  counters := socket_counters(),
  num_readers := integer() >= 0,
  num_writers := integer() >= 0,
  num_acceptors := integer() >= 0,
  writable := boolean(),
  readable := boolean()}

ip4_address() = {0..255, 0..255, 0..255, 0..255}

ip6_address() =
{0..65535, 0..65535, 0..65535, 0..65535, 0..65535, 0..65535, 0..65535, 0..65535}

sockaddr() =
  sockaddr_in4() |
  sockaddr_in6() |
  sockaddr_un() |
  sockaddr_ll()

sockaddr_in4() =
#{family := inet,
  port := port_number(),
  addr := any | broadcast | loopback | ip4_address()}

sockaddr_in6() =
#{family := inet6,
  port := port_number(),
  addr := any | loopback | ip6_address(),
flowinfo := in6_flow_info(),
scope_id := in6_scope_id()

sockaddr_un() = #{family := local, path := binary() | string()}

sockaddr_ll() =
#{family := packet,
  protocol := integer() >= 0,
  ifindex := integer(),
  pkttype := packet_type(),
  htype := integer() >= 0,
  addr := binary()}

packet_type() =
  host | broadcast | multicast | otherhost | outgoing |
  loopback | user | kernel | fastroute |
  integer() >= 0

port_number() = 0..65535

in6_flow_info() = uint20()
in6_scope_id() = uint32()

send_flags() = [send_flag()]
send_flag() = confirm | dontroute | eor | more | nosignal | oob

recv_flags() = [recv_flag()]
recv_flag() = cmsg_cloexec | errqueue | oob | peek | trunc

shutdown_how() = read | write | read_write

sockopt_level() =
  otp | socket | ip | ipv6 | tcp | udp | sctp |
  integer() >= 0

otp_socket_option() =
  debug | iow | controlling_process | rcvbuf | rcvctrlbuf |
  sndctrlbuf | meta | fd

socket_option() =
  acceptconn | acceptfilter | bindtodevice | broadcast |
  busy_poll | debug | domain | dontroute | error | keepalive |
  linger | mark | oobinline | passcred | peek_off | peercred |
  priority | protocol | rcvbuf | rcvbufforce | rcvlowat |
  rcvtimeo | reuseaddr | reuseport | rxq_ovfl | setfib |
  sndbuf | sndbufsize | sndlowat | sndtimeo | timestamp | type

ip_socket_option() =
  add_membership | add_source_membership | block_source |
  dontfrag | drop_membership | drop_source_membership |
  freebind | hdrincl | minttl | msfilter | mtu | mtu_discover |
  multicast_all | multicast_if | multicast_loop |
  multicast_ttl | nodefrag | options | pktinfo | recverr |
  recvif | recvdstaddr | recvopts | recvoridstaddr | recvvtos |
  recvvttl | retopts | router_alert | sndsrcaddr | tos |
  transparent | ttl | unblock_source

ipv6_socket_option() =
  addMembership | authhdr | auth_level | checksum |
  drop_membership | dstopts | esp_trans_level |
  esp_network_level | faith | flowinfo | hopopts |
  ipcomp_level | initial | leave_group | mtu | mtu_discover |
socket

multicast_hops | multicast_if | multicast_loop | portrange |
pktoptions | recverr | recvholimit | hoplimit | recvpktpktinfo |
pktinfo | recvtkclass | router_alert | rthdr | tclass |
unicast_hops | use_m_mtu | v6only

tcp_socket_option() =
  congestion | cork | info | keepcnt | keepidle | keepintvl |
  maxseg | md5sig | nodelay | noopt | nopush | syncnt |
  user_timeout

udp_socket_option() = cork

scpt_socket_option() =
  adaption_layer | associnfo | auth_active_key | auth_asconf |
  auth_chunk | auth_key | auth_delete_key | autoclose |
  context | default_send_params | delayed_ack_time |
  disable_fragments | hmac_ident | events | explicit_eor |
  fragment_interleave | get_peer_addr_info | initmsg |
  i_want_mapped_v4_addr | local_auth_chunks | maxseg |
  maxburst | nodelay | partial_delivery_point |
  peer_addr_params | peer_auth_chunks | primary_addr |
  reset_streams | rtoinfo | set_peer_primary_addr | status |
  use_ext_recvinfo

timeval() = #{sec := integer(), usec := integer()}

ip_tos() =
  lowdelay | throughput | reliability | mincost | integer()

ip_mreq() =
  #{multiaddr := ip4_address(),
    interface := any | ip4_address()}

ip_mreq_source() =
  #{multiaddr := ip4_address(),
    interface := ip4_address(),
    sourceaddr := ip4_address()}

ip_pmtudisc() = want | dont | do | probe

ip_msfilter_mode() = include | exclude

ip_msfilter() =
  #{multiaddr := ip4_address(),
    interface := ip4_address(),
    mode := ip_msfilter_mode(),
    slist := [ip4_address()]}

ip_pktinfo() =
  #{ifindex := integer() >= 0,
    spec_dst := ip4_address(),
    addr := ip4_address()}

ipv6_mreq() =
  #{multiaddr := ip6_address(), interface := integer() >= 0}

ipv6_pmtudisc() = ip_pmtudisc()

ipv6_pktinfo() = #{addr := ip6_address(), ifindex := integer()}

scpt_assoc_id() = int32()

scpt_sndrcvinfo() =
  #{stream := uint16(),
    ssn := uint16(),
    sresid := uint16()}
flags := uint16(),
ppid := uint16(),
context := uint16(),
timetolive := uint16(),
 tsn := uint16(),
cumtsn := uint16(),
assoc_id := sctp_assoc_id()}
sctp_event_subscribe() =
#{data_in := boolean(),
association := boolean(),
address := boolean(),
send_failure := boolean(),
 peer_error := boolean(),
 shutdown := boolean(),
 partial_delivery := boolean(),
adaptation_layer := boolean(),
authentication := boolean(),
sender_dry := boolean()}
sctp_assocparams() =
#{assoc_id := sctp_assoc_id(),
max_rxt := uint16(),
num_peer_dests := uint16(),
 peer_rwnd := uint32(),
local_rwnd := uint32(),
 cookie_life := uint32()}
sctp_initmsg() =
#{num_outstreams := uint16(),
max_instreams := uint16(),
max_attempts := uint16(),
max_init_timeo := uint16()}
sctp_rtoinfo() =
#{assoc_id := sctp_assoc_id(),
initial := uint32(),
 max := uint32(),
 min := uint32()}
msghdr_flag() = ctrunc | eor | errqueue | oob | trunc
msghdr_flags() = [msghdr_flag()]
msghdr() =
#{addr := sockaddr(),
iov := [binary()],
ctrl := [cmsghdr_recv()] | [cmsghdr_send()],
flags := msghdr_flags()}
cmsghdr_level() = socket | ip | ipv6 | integer()
cmsghdr_type() =
credentials | hoplevel | origdstaddr | pktinfo | recvtos |
rights | timestamp | tos | ttl |
too | integer()
cmsghdr_recv() =
#{level := socket, type := timestamp, data := timeval()} |
#{level := socket, type := rights, data := binary()} |
socket

#{level := socket, type := credentials, data := binary()} |
#{level := socket, type := integer(), data := binary()} |
#{level := ip, type := tos, data := ip_tos()} |
#{level := ip, type := recvtos, data := ip_tos()} |
#{level := ip, type := ttl, data := integer()} |
#{level := ip, type := recvttl, data := integer()} |
#{level := ip, type := pktinfo, data := ip_pktinfo()} |
#{level := ip, type := origdstaddr, data := sockaddr_in4()} |
#{level := ip, type := recverr, data := extended_err() | binary()} |
#{level := ip, type := integer(), data := binary()} |
#{level := ipv6, type := hoplevel, data := integer()} |
#{level := ipv6, type := pktinfo, data := ipv6_pktinfo()} |
#{level := ipv6, type := recverr, data := extended_err() | binary()} |
#{level := ipv6, type := tclass, data := integer()} |
#{level := ipv6, type := integer(), data := binary()} |
#{level := integer(), type := integer(), data := binary()}

cmsghdr_send() =
#{level := socket, type := timestamp, data := binary()} |
#{level := socket, type := rights, data := binary()} |
#{level := socket, type := credentials, data := binary()} |
#{level := ip, type := tos, data := ip_tos()} |
#{level := ip, type := integer(), data := binary()} |
#{level := ip, type := integer(), data := binary()} |
#{level := ip, type := integer(), data := binary()} |
#{level := ipv6, type := tclass, data := integer()} |
#{level := ipv6, type := integer(), data := binary()} |
#{level := udp, type := integer(), data := binary()}

icmp_dest_unreach() =
net_unreach | host_unreach | port_unreach | frag_needed | net_unknown | host_unknown |
uint8()

icmpv6_dest_unreach() =
noroute | adm_prohibited | not_neighbour | addr_unreach | port_unreach | policy_fail | reject_route |
uint8()

ee_origin() = none | local | icmp | icmp6 | uint8()

extended_err() =
#{error := term(), origin := icmp, type := dest_unreach, code := icmp_dest_unreach(), info := uint32(), data := uint32(), offender := undefined | sockaddr()} |
#{error := term(), origin := icmp,}
type := time_exceeded | uint8(),
code := uint8(),
info := uint32(),
data := uint32(),
offender := undefined | sockaddr()
#{error := term(),
    origin := icmp6,
    type := dest_unreach,
    code := icmpv6_dest_unreach(),
    info := uint32(),
    data := uint32(),
    offender := undefined | sockaddr()}
#{error := term(),
    origin := icmp6,
    type := pkt_toobig | time_exceeded | uint8(),
    code := uint8(),
    info := uint32(),
    data := uint32(),
    offender := undefined | sockaddr()}
#{error := term(),
    origin := ee_origin(),
    type := uint8(),
    code := uint8(),
    info := uint32(),
    data := uint32(),
    offender := undefined | sockaddr()}

uint8() = 0..255
uint16() = 0..65535
uint20() = 0..1048575
uint32() = 0..4294967295
int32() = -2147483648..2147483647
errcode() =
    inet:posix() | exalloc | exmonitor | exselect | exself

The POSIX error codes are mostly come from the OS level socket interface, but this module may generate some appropriate POSIX codes.
The other values come from this module's lower levels and are all fairly fatal internal errors:

exalloc
    Memory allocation failed
exmonitor
    Failed to set a monitor on a process
exselect
    Select operation failed
exself
    Failed to get current process
Exports

accept(LSocket) -> {ok, Socket} | {error, Reason}
accept(LSocket, Timeout) -> {ok, Socket} | {error, Reason}
Types:
    LSocket = socket()
    Timeout = timeout()
    Socket = socket()
    Reason = errcode() | closed | timeout

Accept a connection on a socket.
This call is used with connection-based socket types (stream or seqpacket). It extracts the first pending connection request for the listen socket and returns the (newly) connected socket.

accept(LSocket, Timeout :: nowait) ->
    {ok, Socket} | {select, SelectInfo} | {error, Reason}
Types:
    LSocket = Socket = socket()
    SelectInfo = select_info()
    Reason = errcode() | closed

Accept a connection on a socket.
This call is used with connection-based socket types (stream or seqpacket). It extracts the first pending connection request for the listen socket and returns the (newly) connected socket.
In the case when there is no connections waiting, the function will return with the SelectInfo. The caller can then await a select message, {'$socket', Socket, select, Info} (where Info is the ref field from the SelectInfo), when a client connects (a subsequent call to accept will then return the socket).

bind(Socket, Addr) -> {ok, Port} | {error, Reason}
Types:
    Socket = socket()
    Addr = sockaddr() | any | broadcast | loopback
    Port = port_number()
    Reason = inet:posix() | closed

Bind a name to a socket.
When a socket is created (with open), it has no address assigned to it. bind assigns the address specified by the Addr argument.
The rules used for name binding vary between domains.

cancel(Socket, SelectInfo) -> ok | {error, Reason}
Types:
    Socket = socket()
    SelectInfo = select_info()
    Reason = einval | closed | exself

Cancel an asynchronous request.
Call this function in order to cancel a previous asynchronous call to, e.g. `recv/3`.

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

**Types:**
- `Socket` = `socket()`
- `Reason` = `endcode() | closed | timeout`

Closes the socket.

**Note:**

Note that for e.g. `protocol = tcp`, most implementations doing a close does not guarantee that any data sent is delivered to the recipient before the close is detected at the remote side.

One way to handle this is to use the `shutdown` function (`socket:shutdown(Socket, write)`) to signal that no more data is to be sent and then wait for the read side of the socket to be closed.

```
connect(Socket, SockAddr) -> ok | {error, Reason}
connect(Socket, SockAddr, Timeout) -> ok | {error, Reason}
```

**Types:**
- `Socket` = `socket()`
- `SockAddr` = `sockaddr()`
- `Timeout` = `timeout()`
- `Reason` = `endcode() | closed | timeout`

This function connects the socket to the address specified by the `SockAddr` argument.

```
connect(Socket, SockAddr, Timeout :: nowait) ->
  ok | {select, SelectInfo} | {error, Reason}
```

**Types:**
- `Socket` = `socket()`
- `SockAddr` = `sockaddr()`
- `SelectInfo` = `select_info()`
- `Reason` = `endcode() | closed`

This function connects the socket to the address specified by the `SockAddr` argument.

In the case when it's not possible to immediately establish a connection, the function will return with the `SelectInfo`. The caller can then await a select message, `{'$socket', Socket, select, Info}` (where `Info` is the ref field from the `SelectInfo`, a subsequent call to connect will then establish the connection).

```
getopt(Socket, Level :: otp, Key :: otp_socket_option()) ->
  {ok, Value} | {error, Reason}
getopt(Socket, Level :: socket, Key :: socket_option()) ->
  {ok, Value} | {error, Reason}
getopt(Socket, Level :: ip, Key :: ip_socket_option()) ->
  {ok, Value} | {error, Reason}
getopt(Socket, Level :: ipv6, Key :: ipv6_socket_option()) ->
  {ok, Value} | {error, Reason}
getopt(Socket, Level :: tcp, Key :: tcp_socket_option()) ->
```

Ericsson AB. All Rights Reserved.: Kernel | 315
getopt(Socket, Level :: udp, Key :: udp_socket_option()) ->
{ok, Value} | {error, Reason}
getopt(Socket, Level :: sctp, Key :: sctp_socket_option()) ->
{ok, Value} | {error, Reason}

Types:

Socket = socket()
Value = term()
Reason = inet:posix() | closed

Get an option on a socket.

What properties are valid depend both on Level and on what kind of socket it is (domain, type and protocol).
See the socket options chapter of the users guide for more info.

**Note:**

Not all options are valid on all platforms. That is, even if "we" support an option, that does not mean that the underlying OS does.

getopt(Socket, Level, Key) -> ok | {ok, Value} | {error, Reason}

Types:

Socket = socket()
Level = integer()
Key = {NativeOpt, ValueSize}
NativeOpt = integer()
ValueSize = int | bool | integer() >= 0
Value = term()
Reason = inet:posix() | closed

Get an option on a socket.

When specifying Level as an integer, and therefor using "native mode", it is currently up to the caller to know how to interpret the result.
For more info, see getopt above.

info(Socket) -> socket_info()

Types:

Socket = socket()

Get miscellaneous info about the socket.

The function returns a map with each info item as a key-value binding. It reflects the "current" state of the socket.

**Note:**

In order to ensure data integrity, mutex'es are taken when needed. So, do not call this function often.
listen(Socket) -> ok | {error, Reason}
listen(Socket, Backlog) -> ok | {error, Reason}

Types:
   Socket = socket()
   Backlog = integer()
   Reason = inet:posix() | closed

Listen for connections on a socket.

number_of() -> integer() >= 0

Returns the number of active sockets.

open(FD) -> {ok, Socket} | {error, Reason}
open(FD, Opts) -> {ok, Socket} | {error, Reason}

Types:
   FD = integer()
   Opts =
     #{domain => domain(),
       type => type(),
       protocol => protocol(),
       dup => boolean()}
   Socket = socket()
   Reason = errcode()

Create an endpoint (socket) for communication based on an already existing file descriptor. The function attempts to retrieve domain, type and protocol from the system. This is however not possible on all platforms, and in those cases it expects it in Opts.

The Opts argument is intended for providing extra information for the open call:

dup: boolean()
   Shall the provided descriptor be duplicated (dup) or not.
   Defaults to true.

debug: boolean()
   Enable or disable debug during the open call.
   Defaults to false.

domain: socket:domain()
   Which domain is the descriptor of.

type: socket:type()
   Which type is the descriptor of.

protocol: socket:protocol()
   Which protocol is the descriptor of.
Note:
This function should be used with care!
On some platforms its necessary to provide the protocol as its impossible to retrieve it.

```erlang
open(Domain, Type) -> {ok, Socket} | {error, Reason}
open(Domain, Type, Protocol) -> {ok, Socket} | {error, Reason}
open(Domain, Type, Protocol, Opts) ->
    {ok, Socket} | {error, Reason}
```

Types:
- **Domain** = domain()
- **Type** = type()
- **Protocol** = default | protocol()
- **Opts** = map()
- **Socket** = socket()
- **Reason** = errcode()

Creates an endpoint (socket) for communication.

For some types there is a default protocol, indicated by default, which it may be possible to specify. And for Domain = local, if a protocol is specified, it must be default.

The Opts argument is intended for "other" options. Currently the only supported option(s) are netns, which is only supported on the linux platform and debug (controls debug printouts during the open call).

Note:
It may not be possible to specify the default protocol (except when Domain = local). We need to be able to retrieve the resulting protocol, which is not possible on all platforms.

```erlang
peername(Socket) -> {ok, SockAddr} | {error, Reason}
```

Types:
- **Socket** = socket()
- **SockAddr** = sockaddr()
- **Reason** = inet:posix() | closed

Returns the address of the peer connected to the socket.

```erlang
recv(Socket) -> {ok, Data} | {error, Reason}
recv(Socket, Length) -> {ok, Data} | {error, Reason}
recv(Socket, Length, Flags) -> {ok, Data} | {error, Reason}
recv(Socket, Length, Timeout) -> {ok, Data} | {error, Reason}
recv(Socket, Length, Flags, Timeout) ->
    {ok, Data} | {error, Reason}
```

Types:
Receive a message from a socket.

There is a special case for the argument Length. If it is set to zero (0), it means "give me everything you currently have".

recv(Socket, Length, Timeout :: nowait) ->
    {ok, Data} |
    {ok, {Data, SelectInfo}} |
    {select, SelectInfo} |
    {error, Reason}
recv(Socket, Length, Flags, Timeout :: nowait) ->
    {ok, Data} |
    {ok, {Data, SelectInfo}} |
    {select, SelectInfo} |
    {error, Reason}

Types:
Socket = socket()
Length = integer() >= 0
Flags = recv_flags()
Data = binary()
SelectInfo = select_info()
Reason =
    errcode() | closed | timeout |
    {errcode() | closed | timeout, Data :: binary()}

Receive a message from a socket.

There is a special case for the argument Length. If it is set to zero (0), it means "give me everything you currently have".

In the case when there is no data waiting, the function will return with the SelectInfo. The caller can then await a select message, {socket, Socket, select, Info} (where Info is the ref field from the SelectInfo), when data has arrived (a subsequent call to recv will then return the data).

Note that if a length (> 0) is specified, and only part of that amount of data is available, the function will return with that data and the SelectInfo (if the caller don't want to wait for the remaining data, it must immediately call the cancel/2 function.)

recvfrom(Socket) -> {ok, {Source, Data}} | {error, Reason}
recvfrom(Socket, BufSz) -> {ok, {Source, Data}} | {error, Reason}
recvfrom(Socket, Flags, Timeout) ->
    {ok, {Source, Data}} | {error, Reason}
recvfrom(Socket, BufSz, Flags) ->
recvfrom(Socket, BufSz, Timeout) ->
{ok, {Source, Data}} | {error, Reason}
recvfrom(Socket, BufSz, Flags, Timeout) ->
{ok, {Source, Data}} | {error, Reason}

Types:
Socket = socket()
BufSz = integer() >= 0
Flags = recv_flags()
Timeout = timeout()
Source = sockaddr() | undefined
Data = binary()
Reason = errcode() | closed | timeout

Receive a message from a socket.

This function reads "messages", which means that regardless of how much we want to read, it returns when we get a message (if the buffer size is too small, the message will be truncated).

The BufSz argument basically defines the size of the receive buffer. By setting the value to zero (0), the configured size (setopt with Level = otp and Key = rcvbuf) is used.

It may be impossible to know what (buffer) size is appropriate "in advance", and in those cases it may be convenient to use the (recv) 'peek' flag. When this flag is provided, the message is *not* "consumed" from the underlying buffers, so another recvfrom call is needed, possibly with a then adjusted buffer size.

recvfrom(Socket, Flags, Timeout :: nowait) ->
{ok, {Source, Data}} |
{select, SelectInfo} |
{error, Reason}
recvfrom(Socket, BufSz, Timeout :: nowait) ->
{ok, {Source, Data}} |
{select, SelectInfo} |
{error, Reason}
recvfrom(Socket, BufSz, Flags, Timeout :: nowait) ->
{ok, {Source, Data}} |
{select, SelectInfo} |
{error, Reason}

Types:
Socket = socket()
BufSz = integer() >= 0
Flags = recv_flags()
Source = sockaddr() | undefined
Data = binary()
SelectInfo = select_info()
Reason = errcode() | closed

Receive a message from a socket.

This function reads "messages", which means that regardless of how much we want to read, it returns when we get a message (if the buffer size is too small, the message will be truncated).
The BufSz argument basically defines the size of the receive buffer. By setting the value to zero (0), the configured size (setopt with Level = otp and Key = rcvbuf) is used.

It may be impossible to know what (buffer) size is appropriate "in advance", and in those cases it may be convenient to use the (recv) 'peek' flag. When this flag is provided, the message is *not* "consumed" from the underlying buffers, so another recvfrom call is needed, possibly with a then adjusted buffer size.

In the case when there is no data waiting, the function will return with the SelectInfo. The caller can then await a select message, ('$socket', Socket, select, Info) (where Info is the ref field from the SelectInfo), when data has arrived (a subsequent call to recvfrom will then return the data).

recvmsg(Socket) -> {ok, MsgHdr} | {error, Reason}
recvmsg(Socket, Flags) -> {ok, MsgHdr} | {error, Reason}
recvmsg(Socket, Timeout) -> {ok, MsgHdr} | {error, Reason}
recvmsg(Socket, Flags, Timeout) -> {ok, MsgHdr} | {error, Reason}
recvmsg(Socket, BufSz, CtrlSz) -> {ok, MsgHdr} | {error, Reason}
recvmsg(Socket, BufSz, CtrlSz, Flags, Timeout) -> {ok, MsgHdr} | {error, Reason}

Types:
  Socket = socket()
  BufSz = CtrlSz = integer() >= 0
  Flags = recv_flags()
  Timeout = timeout()
  MsgHdr = msghdr()
  Reason = errcode() | closed | timeout

Receive a message from a socket.

This function reads "messages", which means that regardless of how much we want to read, it returns when we get a message.

The message will be delivered in the form of a msghdr(), which may contain the source address (if socket not connected), a list of cmsghdr_recv() (depends on what socket options have been set and what the protocol and platform supports) and also a set of flags, providing further info about the read.

The BufSz argument basically defines the size of the receive buffer. By setting the value to zero (0), the configured size (setopt with Level = otp and Key = rcvbuf) is used.

The CtrlSz argument basically defines the size of the receive buffer for the control messages. By setting the value to zero (0), the configured size (setopt with Level = otp) is used.

It may be impossible to know what (buffer) size is appropriate "in advance", and in those cases it may be convenient to use the (recv) 'peek' flag. When this flag is provided, the message is *not* "consumed" from the underlying buffers, so another recvmsg call is needed, possibly with a then adjusted buffer size.

recvmsg(Socket, Timeout :: nowait) -> {ok, MsgHdr} | {select, SelectInfo} | {error, Reason}
recvmsg(Socket, Flags, Timeout :: nowait) -> {ok, MsgHdr} | {select, SelectInfo} | {error, Reason}
recvmsg(Socket, BufSz, CtrlSz, Flags, Timeout :: nowait) -> {ok, MsgHdr} | {select, SelectInfo} | {error, Reason}

Types:
Receive a message from a socket.

This function reads "messages", which means that regardless of how much we want to read, it returns when we get a message.

The message will be delivered in the form of a `msghdr()`, which may contain the source address (if socket not connected), a list of `cmsghdr_recv()` (depends on what socket options have been set and what the protocol and platform supports) and also a set of flags, providing further info about the read.

The `BufSz` argument basically defines the size of the receive buffer. By setting the value to zero (0), the configured size (setopt with `Level = otp` and `Key = rcvbuf`) is used.

The `CtrlSz` argument basically defines the size of the receive buffer for the control messages. By setting the value to zero (0), the configured size (setopt with `Level = otp`) is used.

It may be impossible to know what (buffer) size is appropriate "in advance", and in those cases it may be convenient to use the (recv) 'peek' flag. When this flag is provided, the message is *not* "consumed" from the underlying buffers, so another recvmsg call is needed, possibly with a then adjusted buffer size.

In the case when there is no data waiting, the function will return with the `SelectInfo`. The caller can then await a select message, `{'$socket', Socket, select, Info}` (where `Info` is the ref field from the `SelectInfo`), when data has arrived (a subsequent call to recvmsg will then return the data).

```
send(Socket, Data) -> ok | {error, Reason}
send(Socket, Data, Flags) -> ok | {error, Reason}
send(Socket, Data, Timeout) -> ok | {error, Reason}
send(Socket, Data, Flags, Timeout) -> ok | {error, Reason}
```

Types:
- `Socket = socket()`
- `Data = iodata()`
- `Flags = send_flags()`
- `Timeout = timeout()`
- `Reason = {errcode() | closed | timeout, Remaining :: integer() >= 1}`

Send a message on a connected socket.

```
send(Socket, Data, Timeout :: nowait) ->
    ok |
    {ok, {binary(), SelectInfo}} |
    {select, SelectInfo} |
    {ok, {RestData, SelectInfo}} |
    {error, Reason}

send(Socket, Data, Flags, Timeout :: nowait) ->
    ok |
    {select, SelectInfo} |
```
Send a message on a connected socket.

In the case when there is no room in the (system-) buffers, the function will return with the SelectInfo. The caller can then await a select message, \{"$socket", Socket, select, Info\} (where Info is the ref field from the SelectInfo), when there is room for more data (a subsequent call to send will then send the data).

Note that if not all the data was sent, the function will return with the remaining data and the SelectInfo (if the caller don't want to wait to be able to send the rest, it should immediately call the cancel/2 function.)

```erlang
sendmsg(Socket, MsgHdr) -> ok | {ok, Remaining} | {error, Reason}
sendmsg(Socket, MsgHdr, Flags) -> ok | {error, Reason}
sendmsg(Socket, MsgHdr, Timeout) -> ok | {error, Reason}
sendmsg(Socket, MsgHdr, Flags, Timeout) ->
    ok | {ok, Remaining} | {error, Reason}
```

Types:

```
Socket = socket()
MsgHdr = msghdr()
Flags = send_flags()
Timeout = timeout()
Remaining = erlang:iovec()
Reason = {errcode() | closed, Remaining :: integer() >= 1}
```

Send a message on a socket. The destination, if needed (socket not connected) is provided in the MsgHdr, which also contains the message to send. The MsgHdr may also contain an list of optional cmsghdr_send() (depends on what the protocol and platform supports).

Unlike the send function, this one sends one message. This means that if, for whatever reason, its not possible to send the message in one go, the function will instead return with the remaining data({ok, Remaining}). Thereby leaving it up to the caller to decide what to do (retry with the remaining data of give up).

```erlang
sendmsg(Socket, MsgHdr, Timeout :: nowait) ->
    ok | {ok, Remaining} | {error, Reason}
sendmsg(Socket, MsgHdr, Flags, Timeout :: nowait) ->
    ok | {ok, Remaining} | {error, Reason}
```

Types:
Socket = socket()
MsgHdr = msghdr()
Flags = send_flags()
Remaining = erlang:iowvec()
Reason = errcode() | closed

Send a message on a socket. The destination, if needed (socket not connected) is provided in the MsgHdr, which also contains the message to send. The MsgHdr may also contain an list of optional cmsghdr_send() (depends on what the protocol and platform supports).

Unlike the send function, this one sends one message. This means that if, for whatever reason, its not possible to send the message in one go, the function will instead return with the remaining data ({ok, Remaining}). Thereby leaving it up to the caller to decide what to do (retry with the remaining data of give up).

In the case when there is no room in the (system-) buffers, the function will return with the SelectInfo. The caller can then await a select message, {'$socket', Socket, select, Info} (where Info is the ref field from the SelectInfo), when there is room for more data (a subsequent call to sendmsg will then send the data).

sendto(Socket, Data, Dest) -> ok | {error, Reason}
sendto(Socket, Data, Dest, Flags) -> ok | {error, Reason}
sendto(Socket, Data, Dest, Timeout) -> ok | {error, Reason}
sendto(Socket, Data, Dest, Flags, Timeout) -> ok | {error, Reason}

Types:
Socket = socket()
Data = binary()
Dest = sockaddr()
Flags = send_flags()
Timeout = timeout()
Reason =
    {errcode() | closed | timeout, Remaining :: integer() >= 1}

Send a message on a socket, to the specified destination.

sendto(Socket, Data, Dest, Timeout :: nowait) ->
    ok | {select, SelectInfo} | {error, Reason}
sendto(Socket, Data, Dest, Flags, Timeout :: nowait) ->
    ok |
        {ok, {binary(), SelectInfo}} |
        {select, SelectInfo} |
        {error, Reason}

Types:
Socket = socket()
Data = binary()
Dest = sockaddr()
Flags = send_flags()
SelectInfo = select_info()
Reason = {errcode() | closed, Remaining :: integer() >= 1}

Send a message on a socket, to the specified destination.
In the case when there is no room in the (system-) buffers, the function will return with the `SelectInfo`. The caller can then await a select message, `{socket, Socket, select, Info}` (where `Info` is the ref field from the `SelectInfo`), when there is room for more data (a subsequent call to `sendto` will then send the data).

```erlang
setopt(Socket, Level :: otp, Key :: otp_socket_option(), Value) ->
  ok | {error, Reason}
setopt(Socket, Level :: socket, Key :: socket_option(), Value) ->
  ok | {error, Reason}
setopt(Socket, Level :: ip, Key :: ip_socket_option(), Value) ->
  ok | {error, Reason}
setopt(Socket, Level :: ipv6, Key :: ipv6_socket_option(), Value) ->
  ok | {error, Reason}
setopt(Socket, Level :: tcp, Key :: tcp_socket_option(), Value) ->
  ok | {error, Reason}
setopt(Socket, Level :: udp, Key :: udp_socket_option(), Value) ->
  ok | {error, Reason}
setopt(Socket, Level :: scpt, Key :: scpt_socket_option(), Value) ->
  ok | {error, Reason}
setopt(Socket, Level, Key, Value) -> ok | {error, Reason}
```

**Types:**
- `Socket = socket()
- `Level = Key = integer() >= 0
- `Value = binary()
- `Reason = inet:posix() | closed

Set an option on a socket.

What options are valid depend both on `Level` and on what kind of socket it is (domain, type and protocol). See the socket options chapter of the users guide for more info.

**Note:**

Not all options are valid on all platforms. That is, even if "we" support an option, that does not mean that the underlying OS does.

**Note:**

Sockets are set 'non-blocking' when created, so this option is *not* available (as it would adversely effect the Erlang VM to set a socket 'blocking').

```erlang
setopt(Socket, Level, Key, Value) -> ok | {error, Reason}
```

**Types:**
- `Socket = socket()
- `Level = Key = integer() >= 0
- `Value = binary()
- `Reason = inet:posix() | closed

Set options on a socket.
When specifying Level as an integer, and thereafter using "native mode", it is *currently* up to the caller to know how to encode the Value.

For more info, see setopt above.

**shutdown** (Socket, How) -> ok | {error, Reason}

*Types:*

- Socket = socket()
- How = shutdown_how()
- Reason = inet:posix() | closed

Shut down all or part of a full-duplex connection.

**sockname** (Socket) -> {ok, SockAddr} | {error, Reason}

*Types:*

- Socket = socket()
- SockAddr = sockaddr()
- Reason = inet:posix() | closed

Returns the current address to which the socket is bound.

**supports()** -> Supports

**supports(Key1 :: options)** -> SupportsOptions

**supports(Key1 :: send_flags)** -> SupportsSendFlags

**supports(Key1 :: recv_flags)** -> SupportsRecvFlags

**supports(Key1 :: options, Key2 :: socket)** -> SupportsOptionsSocket

**supports(Key1 :: options, Key2 :: ip)** -> SupportsOptionsIP

**supports(Key1 :: options, Key2 :: ipv6)** -> SupportsOptionsIPv6

**supports(Key1 :: options, Key2 :: tcp)** -> SupportsOptionsTCP

**supports(Key1 :: options, Key2 :: udp)** -> SupportsOptionsUDP

**supports(Key1 :: options, Key2 :: sctp)** -> SupportsOptionsSCTP

*Types:*

- Supports :: [{Feature, boolean()} | {send_flags, SupportsSendFlags} | {recv_flags, SupportsRecvFlags} | {options, SupportsOptions}]
- Feature :: sctp | ipv6 | local | netns
- SupportsSendFlags :: [{send_flag(), boolean()}]
- SupportsRecvFlags :: [{recv_flag(), boolean()}]
- SupportsOptions :: [{socket, SupportsOptionsSocket} | {ip, SupportsOptionsIP} | {ipv6, SupportsOptionsIPv6} | {tcp, SupportsOptionsTCP} | {udp, SupportsOptionsUDP} | {sctp, SupportsOptionsSCTP}]
- SupportsOptionsSocket :: [{socket_option(), boolean()}]
- SupportsOptionsIP :: [{ip_socket_option(), boolean()}]
- SupportsOptionsIPv6 :: [{ipv6_socket_option(), boolean()}]
- SupportsOptionsTCP :: [{tcp_socket_option(), boolean()}]
- SupportsOptionsUDP :: [{udp_socket_option(), boolean()}]
- SupportsOptionsSCTP :: [{sctp_socket_option(), boolean()}]
This function retrieves information about what the platform supports, such as if SCTP is supported, or which socket options are supported.

For keys other than the known the empty list is returned. Note that in a future version or on a different platform there might be more supported items.

\[
\begin{align*}
is\_supported&(Key1 :: sctp | ipv6 | local | netns) \rightarrow boolean() \\
is\_supported&(Key1 :: send\_flags, Key2 :: SendFlag) \rightarrow boolean() \\
is\_supported&(Key1 :: recv\_flags, Key2 :: RecvFlag) \rightarrow boolean() \\
is\_supported&(Key1 :: options, Key2 :: socket, Key3 :: SocketOption) \rightarrow boolean() \\
is\_supported&(Key1 :: options, Key2 :: ip, Key3 :: IPSocketOption) \rightarrow boolean() \\
is\_supported&(Key1 :: options, Key2 :: ipv6, Key3 :: IPv6SocketOption) \rightarrow boolean() \\
is\_supported&(Key1 :: options, Key2 :: tcp, Key3 :: TCPSocketOption) \rightarrow boolean() \\
is\_supported&(Key1 :: options, Key2 :: udp, Key3 :: UDPSocketOption) \rightarrow boolean() \\
is\_supported&(Key1 :: options, Key2 :: sctp, Key3 :: SCTPSocketOption) \rightarrow boolean() \\
\end{align*}
\]

Types:

\[
\begin{align*}
SocketOption & :: socket\_option() \\
IPSocketOption & :: ip\_socket\_option() \\
IPv6SocketOption & :: ipv6\_socket\_option() \\
TCPSocketOption & :: tcp\_socket\_option() \\
UDPSocketOption & :: udp\_socket\_option() \\
SCTPSocketOption & :: sctp\_socket\_option() \\
\end{align*}
\]

This function retrieves information about what the platform supports, such as if SCTP is supported, or which socket options are supported.

For keys other than the known \texttt{false} is returned. Note that in a future version or on a different platform there might be more supported items.

\[
\begin{align*}
\text{which\_sockets}() & \rightarrow [\text{socket()}] \\
\text{which\_sockets}(FilterRule) & \rightarrow [\text{socket()}] \\
\end{align*}
\]

Types:

\[
\begin{align*}
FilterRule & = \\
& \text{inet} | \text{inet6} | \text{stream} | \text{dgram} | \text{seqpacket} | \text{sctp} | \text{tcp} | \text{udp} | \text{pid}() | \;
& \text{fun}(\text{socket\_info()} \rightarrow boolean()) \\
\end{align*}
\]

Returns a list of all sockets, according to the filter rule.

There are several pre-made filter rule(s) and one general:

\[
\begin{align*}
\text{inet} | \text{inet6} \\
\text{Selection based on the domain of the socket.} \\
\text{Only a subset is valid.} \\
\end{align*}
\]
socket

<table>
<thead>
<tr>
<th>stream</th>
<th>dgram</th>
<th>seqpacket</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection based on the type of the socket. Only a subset is valid.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

sctp | tcp | udp

Selection based on the protocol of the socket. Only a subset is valid.

pid()

Selection base on which sockets has this pid as Controlling Process.

fun((socket_info()) -> boolean())

The general filter rule. A fun that takes the socket info and returns a boolean() (true if the socket should be included and false if should not).

Examples

```erlang
client(Addr, SAddr, SPort) ->
  {ok, Sock} = socket:open(inet, stream, tcp),
  {ok, _} = socket:bind(Sock, #{family => inet, addr => Addr}),
  ok = socket:connect(Sock, #{family => inet, addr => Addr, port => SPort}),
  Msg = list_to_binary("hello"),
  ok = socket:send(Sock, Msg),
  ok = socket:shutdown(Sock, write),
  {ok, Msg} = socket:recv(Sock),
  ok = socket:close(Sock).

server(Addr, Port) ->
  {ok, LSock} = socket:open(inet, stream, tcp),
  {ok, _} = socket:bind(LSock, #{family => inet, port => Port, addr => Addr}),
  ok = socket:listen(LSock),
  {ok, Sock} = socket:accept(LSock),
  {ok, Msg} = socket:recv(Sock),
  ok = socket:send(Sock, Msg),
  ok = socket:shutdown(Sock, write),
  ok = socket:close(Sock),
  ok = socket:close(LSock).
```

328 | Ericsson AB. All Rights Reserved.: Kernel
**user**

Erlang module

**user** is a server that responds to all messages defined in the I/O interface. The code in **user.erl** can be used as a model for building alternative I/O servers.
wrap_log_reader

Erlang module

This module makes it possible to read internally formatted wrap disk logs, see disk_log(3). wrap_log_reader does not interfere with disk_log activities; there is however a bug in this version of the wrap_log_reader, see section Known Limitations.

A wrap disk log file consists of many files, called index files. A log file can be opened and closed. Also, a single index file can be opened separately. If a non-existent or non-internally formatted file is opened, an error message is returned. If the file is corrupt, no attempt is made to repair it, but an error message is returned.

If a log is configured to be distributed, it is possible that all items are not logged on all nodes. wrap_log_reader only reads the log on the called node; it is up to the user to be sure that all items are read.

Data Types

continuation()

Continuation returned by open/1,2 or chunk/1,2.

Exports

chunk(Continuation) -> chunk_ret()
chunk(Continuation, N) -> chunk_ret()

Types:

Continuation = continuation()
N = infinity | integer() >= 1
chunk_ret() =

{Continuation2, Terms :: [term()]} |
{Continuation2, Terms :: [term()],
 Badbytes :: integer() >= 0} |
{Continuation2, eof} |
{error, Reason :: term()}

Enables to efficiently read the terms that are appended to a log. Minimises disk I/O by reading 64 kilobyte chunks from the file.

The first time chunk() is called, an initial continuation returned from open/1 or open/2 must be provided.

When chunk/3 is called, N controls the maximum number of terms that are read from the log in each chunk. Defaults to infinity, which means that all the terms contained in the 8K chunk are read. If less than N terms are returned, this does not necessarily mean that end of file is reached.

Returns a tuple {Continuation2, Terms}, where Terms is a list of terms found in the log. Continuation2 is yet another continuation that must be passed on to any subsequent calls to chunk(). With a series of calls to chunk(), it is then possible to extract all terms from a log.

Returns a tuple {Continuation2, Terms, Badbytes} if the log is opened in read only mode and the read chunk is corrupt. Badbytes indicates the number of non-Erlang terms found in the chunk. Notice that the log is not repaired.

Returns {Continuation2, eof} when the end of the log is reached, and {error, Reason} if an error occurs.
The returned continuation either is or is not valid in the next call to this function. This is because the log can wrap and delete the file into which the continuation points. To ensure this does not occur, the log can be blocked during the search.

\[
\text{close(Continuation) -> ok | \{error, Reason\}}
\]

Types:
- \text{Continuation} = \text{continuation()}
- \text{Reason} = \text{file:posix()}

Closes a log file properly.

\[
\text{open(Filename) -> open_ret()}
\]

\[
\text{open(Filename, N) -> open_ret()}
\]

Types:
- \text{Filename} = \text{string() | atom()}
- \text{N} = \text{integer()}

\[
\text{open_ret()} = \begin{cases}
\text{\{ok, Continuation :: continuation()\}} \\
\text{\{error, Reason :: tuple()\}}
\end{cases}
\]

\text{Filename} specifies the name of the file to be read.
\text{N} specifies the index of the file to be read. If \text{N} is omitted, the whole wrap log file is read; if it is specified, only the specified index file is read.

Returns \text{\{ok, Continuation\}} if the log/index file is opened successfully. \text{Continuation} is to be used when chunking or closing the file.

Returns \text{\{error, Reason\}} for all errors.

**Known Limitations**

This version of \text{wrap_log_reader} does not detect if \text{disk_log} wraps to a new index file between a call to \text{wrap_log_reader:open() and the first call to wrap_log_reader:chunk(). If this occurs, the call to chunk() reads the last logged items in the log file, as the opened index file was truncated by disk_log.}

**See Also**

\text{disk_log(3)}
This module is moved to the ERTS application.