

Kernel

Copyright © 1997-2018 Ericsson AB. All Rights Reserved.
Kernel 5.4.3
April 17, 2018

Copyright © 1997-2018 Ericsson AB. All Rights Reserved.
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. Ericsson AB. All Rights Reserved..

April 17, 2018

1 Reference Manual

kernel

Application

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
- Error logging
- Global name service
- Supervision of Erlang/OTP
- Communication with sockets
- Operating system interface

Error Logger Event Handlers

Two standard error logger event handlers are defined in the Kernel application. These are described in *`error_logger(3)`*.

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:set_signal/2`*.

Configuration

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

`browser_cmd = string() | {M,F,A}`

When pressing the **Help** button in a tool such as Debugger, the help text (an HTML file `File`) is by default displayed in a Netscape browser, which is required to be operational. This parameter can be used to change the command for how to display the help text if another browser than Netscape is preferred, or if another platform than Unix or Windows is used.

If set to a string `Command`, the command `"Command File"` is evaluated using *`os:cmd/1`*.

If set to a module-function-args tuple, `{M,F,A}`, the call `apply(M,F,[File|A])` is evaluated.

`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)`*.

`permissions = [Perm]`

Specifies the default permission for applications when they are started. In this parameter:

- `Perm = {ApplName,Bool}`
- `ApplName = atom()`
- `Bool = boolean()`

Permissions are described in *`application:permit/2`*.

`error_logger = Value`

`Value` is one of:

`tty`

Installs the standard event handler, which prints error reports to `stdio`. This is the default option.

`{file, FileName}`

Installs the standard event handler, which prints error reports to file `FileName`, where `FileName` is a string. The file is opened with encoding UTF-8.

`false`

No standard event handler is installed, but the initial, primitive event handler is kept, printing raw event messages to `tty`.

`silent`

Error logging is turned off.

`error_logger_format_depth = Depth`

Can be used to limit the size of the formatted output from the error logger event handlers.

Note:

This configuration parameter was introduced in OTP 18.1 and is experimental. Based on user feedback, it can be changed or improved in future releases, for example, to gain better control over how to limit the size of the formatted output. We have no plans to remove this new feature entirely, unless it turns out to be useless.

`Depth` is a positive integer representing the maximum depth to which terms are printed by the error logger event handlers included in OTP. This configuration parameter is used by the two event handlers defined by the Kernel application and the two event handlers in the SASL application. (If you have implemented your own error handlers, this configuration parameter has no effect on them.)

`Depth` is used as follows: Format strings passed to the event handlers are rewritten. The format controls `~p` and `~w` are replaced with `~P` and `~W`, respectively, and `Depth` is used as the depth parameter. For details, see *`io:format/2`* in STDLIB.

Note:

A reasonable starting value for `Depth` is `30`. We recommend to test crashing various processes in your application, examine the logs from the crashes, and then increase or decrease the value.

`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 `error_logger` messages are generated 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. `TickTime` is specified in seconds. Once every `TickTime/4` second, all connected nodes are ticked (if anything else is written to a node). If nothing is received from another node within the last four tick times, that node is considered to be down. This ensures that nodes that are not responding, for reasons such as hardware errors, are considered to be down.

The time `T`, in which a node that is not responding is detected, is calculated as `MinT < T < MaxT`, where:

```
MinT = TickTime - TickTime / 4
MaxT = TickTime + TickTime / 4
```

`TickTime` defaults to `60` (seconds). Thus, `45 < T < 75` seconds.

Notice that **all** communicating nodes are to have the **same** `TickTime` value specified.

Normally, a terminating node is detected immediately.

`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_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_pg2 = true | false`

Starts the `pg2` server (see *`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`.

`start_timer = true | false`

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

Defaults to `false`.

`shell_history = enabled | disabled`

Specifies whether shell history should be logged to disk between usages of `erl`.

`shell_history_drop = [string()]`

Specific log lines that should not be persisted. For example `["q().", "init:stop()."]` will allow to ignore commands that shut the node down. Defaults to `[]`.

`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.

`shell_history_path = string()`

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

`shutdown_func = {Mod, Func}`

Where:

- `Mod = atom()`
- `Func = atom()`

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

`source_search_rules = [DirRule] | [SuffixRule]`

Where:

- `DirRule = {ObjDirSuffix,SrcDirSuffix}`
- `SuffixRule = {ObjSuffix,SrcSuffix,[DirRule]}`
- `ObjDirSuffix = string()`
- `SrcDirSuffix = string()`
- `ObjSuffix = string()`
- `SrcSuffix = string()`

Specifies a list of rules for use by *`filelib:find_file/2 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 `{"ebin", "src"}`, which are used by `filelib:find_file/2`, or triples specifying separate directory suffix rules depending on file name extensions, for example `[{".beam", ".erl", [{"ebin", "src"}]}`, which are used by `filelib:find_source/2`. Both kinds of rules can be mixed in the list.

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

See Also

app(4), *application(3)*, *code(3)*, *disk_log(3)*, *erl_boot_server(3)*, *erl_ddll(3)*, *error_logger(3)*, *file(3)*, *global(3)*, *global_group(3)*, *heart(3)*, *inet(3)*, *net_kernel(3)*, *os(3)*, *pg2(3)*, *rpc(3)*, *seq_trace(3)*, *user(3)*, *timer(3)*

application

Erlang module

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

```
start_type() =
    normal |
    {takeover, Node :: node()} |
    {failover, Node :: node()}
restart_type() = permanent | transient | temporary
tuple_of(T)
```

A tuple where the elements are of type `T`.

Exports

```
ensure_all_started(Application) -> {ok, Started} | {error, Reason}
ensure_all_started(Application, Type) ->
                      {ok, Started} | {error, Reason}
```

Types:

```
Application = atom()
Type = restart_type()
Started = [atom()]
Reason = term()
```

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()} |
```

```
{vsn, Vsn :: string()} |
{id, Id :: string()} |
{modules, [Module :: module()]} |
{registered, Names :: [Name :: atom()]} |
{applications, [Application :: atom()]} |
{included_applications, [Application :: atom()]} |
{env, [{Par :: atom(), Val :: term()}]} |
{start_phases,
 [{Phase :: atom(), PhaseArgs :: term()}] | undefined} |
{maxT, MaxT :: timeout()} |
{maxP, MaxP :: integer() >= 1 | infinity} |
{mod, Start :: {Module :: module(), StartArgs :: term()}}
```

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.

`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.

`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 is the group leader of all the processes in the application. 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`.

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`.

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.

`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.

Callback Module

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:

`StartType = start_type()`
`StartArgs = term()`
`Pid = pid()`
`State = term()`

This function is called whenever an application is started using `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.

`StartType` defines the type of start:

- `normal` if it is a normal startup.
- `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 `start_phases == undefined`.
- `{takeover,Node}` if the application is distributed and started at the current node because of a takeover from `Node`, either because `takeover/2` has been called or because the current node has higher priority than `Node`.
- `{failover,Node}` if the application is distributed and started at the current node because of a failover from `Node`, and the application specification key `start_phases /= undefined`.

`StartArgs` is the `StartArgs` argument defined by the application specification key `mod`.

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

`Module:start_phase(Phase, StartType, PhaseArgs) -> ok | {error, Reason}`

Types:

`Phase = atom()`

StartType = *start_type()*
PhaseArgs = term()
Pid = pid()
State = state()

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 `start_phases == [{Phase,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 `StartType`, see *`Module:start/2`*.

`Module:prep_stop(State) -> NewState`

Types:

State = NewState = term()

This function is called when an application is about to be stopped, before shutting down the processes of the application.

`State` is the state returned from `Module:start/2`, or `[]` if no state was returned. `NewState` is any term and is passed to `Module:stop/1`.

The function is optional. If it is not defined, the processes are terminated and then `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

Erlang module

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 **embedded** or **interactive** mode. Which one is decided by command-line flag `-mode`:

```
% erl -mode interactive
```

The modes are as follows:

- In embedded mode, all code is loaded during system startup according to the boot script. (Code can also be loaded later by explicitly ordering the code server to do so).
- 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.

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 `$OTPROOT/lib`, where `$OTPROOT` 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:

```
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.

`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
prepared_code()
```

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:

`true`

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() =
        (Filename :: file:filename()) | loaded_ret_atoms()
    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.

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

Types:

```
Module = module()
What = embedded | 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.

`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 can not 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:

```
{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(Modules) ->
{ok, Prepared} | {error, [{Module, What}]}

Types:

Modules = [Module | {Module, Filename, Binary}]
Module = module()
Filename = *file:filename()*
Binary = binary()
Prepared = *prepared_code()*
What = badfile | nofile | on_load_not_allowed | duplicated

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`.

`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 can not 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.

`ensure_modules_loaded(Modules :: [Module]) ->`
`ok | {error, [{Module, What}]}`

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.

`delete(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_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 = *file:filename()* | *loaded_ret_atoms()* | non_existing
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.

If the module is loaded, it returns the name of the file containing the loaded object code.

If the module is preloaded, `preloaded` is returned.

If the module is Cover-compiled, `cover_compiled` is returned.

If the module cannot be found, `non_existing` is returned.

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:

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

root_dir() -> *file:filename()*

Returns the root directory of Erlang/OTP, which is the directory where it is installed.

Example:

```
> code: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:

```
> code: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:

```
> code: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()) ->
                  not_loaded | loaded | modified | removed
```

Returns:

`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`*.

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.

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

```
log() = term()
dlog_size() =
    infinity |
    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`*.

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`, `bchunk/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} |
      eof |
      {error, Reason :: chunk_error_rsn()}
chunk_error_rsn() =
      no_such_log |
      {format_external, log()} |
      {blocked_log, log()} |
      {badarg, continuation} |
      {not_internal_wrap, log()} |
      {corrupt_log_file, FileName :: file:filename()} |
      {file_error, file:filename(), file_error()}
```

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 |
    {format_external, Log} |
    {blocked_log, Log} |
    {badarg, continuation} |
    {file_error, file:filename(), file_error()}
```

Can be used with `chunk/2,3` and `bchunk/2,3` to search through an internally formatted wrap log. It takes as argument a continuation as returned by `chunk/2,3`, `bchunk/2,3`, or `chunk_step/3`, and steps forward (or backward) `Step` files in the wrap log. The continuation returned, points to the first log item in the new current file.

If atom `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 `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 `format_error/1` in module *`file`* is called.

inc_wrap_file(Log) -> ok | {error, *inc_wrap_error_rsn()*}

Types:

```
Log = log()
inc_wrap_error_rsn() =
    no_such_log |
    nonode |
```

```
      {read_only_mode, log()} |
      {blocked_log, log()} |
      {halt_log, log()} |
      {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 `wrap` message, but if an error occurs with a reason tag of `invalid_header` or `file_error`, an `error_status` message is sent.

```
info(Log) -> InfoList | {error, no_such_log}
```

Types:

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

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.

`{no_current_items, integer() >= 0}`

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

`{no_items, integer() >= 0}`

The total number of items in all wrap log files.

`{current_file, integer()}`

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`.

`{no_overflows, {SinceLogWasOpened, SinceLastInfo}}`

`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`.

`lclose(Log) -> ok | {error, lclose_error_rsn()}`
`lclose(Log, Node) -> ok | {error, lclose_error_rsn()}`

Types:

```
Log = log()
Node = node()
lclose_error_rsn() =
    no_such_log | {file_error, file:filename(), file_error()}
```

`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.

`log(Log, Term) -> ok | {error, Reason :: log_error_rsn()}`
`blog(Log, Bytes) -> ok | {error, Reason :: log_error_rsn()}`

Types:

```
Log = log()
Term = term()
Bytes = 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 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, Resaon :: 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()} |
```

```
    {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()} |
    {open_read_only, 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`.

`{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.

`{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 `{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 `{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 `{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 `{{failed,Reason},[{disk_log,open,1}]}`. The function returns `{error, Reason}` for all other errors.

`pid2name(Pid) -> {ok, Log} | undefined`

Types:

`Pid = pid()`
`Log = 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.

`reopen(Log, File) -> ok | {error, reopen_error_rsn()}`
`reopen(Log, File, Head) -> ok | {error, reopen_error_rsn()}`
`breopen(Log, File, BHead) -> ok | {error, reopen_error_rsn()}`

Types:

`Log = log()`
`File = file:filename()`
`Head = term()`
`BHead = iodata()`
`reopen_error_rsn() =`
`no_such_log |`
`nonode |`
`{read_only_mode, log()} |`
`{blocked_log, log()} |`

```
      {same_file_name, log()} |
      {invalid_index_file, file:filename()} |
      {invalid_header, invalid_header()} |
      {file_error, file:filename(), 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 |
    {not_blocked, log()} |
    {not_blocked_by_pid, log()}
```

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:

```
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 getting 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 `.ddl` 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 interrest 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()`
`path() = 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).

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 `{pid(),integer() >= 0}`, where `integer()` denotes the number of users in process `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 `kill_ports`.

`port_count`

Returns the number of ports (an `integer() >= 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 `{pid(),integer() >= 0}`, where `integer()` is the number of monitors held by process `pid()`.

`awaiting_unload`

Returns a list of all processes having monitors for `unloading` active. Each process is returned as `{pid(),integer() >= 0}`, where `integer()` is the number of monitors held by process `pid()`.

If option `linked_in_driver` or `permanent` returns `true`, all other options return `linked_in_driver` or `permanent`, respectively.

The function throws a `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 `Name`. `Path` is a file path to the directory containing the driver. `Name` must be a sharable object/dynamic library. Two drivers with different `Path` parameters cannot be loaded under the same name. `Name` is a string or atom containing at least one character.

The `Name` specified is to correspond to the filename of the dynamically loadable object file residing in the directory specified as `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 `load/2` stops the unloading and keeps the driver (as long as `Path` is the same), and `ok` is returned. If you really want the object code to be reloaded, use *`reload/2`* or the low-level interface *`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 `Path`, or if the same process tries to load it many times, the function returns `ok`. The emulator keeps track of the `load/2` calls, so that a corresponding number of `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 `Path` parameters.

Note:

`Path` is interpreted literally, so that all loaders of the same driver must specify the same **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 `ok`. On failure, the return value is `{error,ErrorDesc}`, where `ErrorDesc` is an opaque term to be translated into human readable form by function *`format_error/1`*.

For more control over the error handling, use the *`try_load/3`* interface instead.

The function throws a `badarg` exception if the parameters are not specified as described here.

`load_driver(Path, Name) -> ok | {error, ErrorDesc}`

Types:

`Path = path()`
`Name = driver()`
`ErrorDesc = term()`

Works essentially as `load/2`, but loads the driver with other options. All ports using the driver are killed with reason `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 *`try_load/3`* with `{driver_options,[kill_ports]}` in the option list gives the same effect regarding the port killing.

The function throws a `badarg` exception if the parameters are not specified as described here.

`loaded_drivers() -> {ok, Drivers}`

Types:

`Drivers = [Driver]`
`Driver = string()`

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 *`info`*.

monitor(Tag, Item) -> MonitorRef

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 **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:

`Tag`

The monitor tag is always `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.

`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:

`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 `DOWN` message sent immediately. Monitoring for loading is therefore most useful when triggered by function *try_load/3*, where the monitor is created **because** the driver is in such a pending state.

Setting a driver monitor for `loading` eventually leads to one of the following messages being sent:

`{'UP', reference(), 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', reference(), 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', reference(), driver, Name, load_cancelled}`

This message arrives if reloading was underway, but the requesting *user* cancelled it by dying or calling *try_unload/2* (or `unload/1`/`unload_driver/1`) again before it was reloaded.

`{'DOWN', reference(), 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 *try_load/3*. The error term can be passed to

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 *`try_load/3`* interface instead.

The function throws a `badarg` exception if the parameters are not specified as described here.

`reload_driver(Path, Name) -> ok | {error, ErrorDesc}`

Types:

```
Path = path()
Name = driver()
ErrorDesc = pending_process | OpaqueError
OpaqueError = term()
```

Works exactly as *`reload/2`*, but for drivers loaded with the *`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 *`scenarios`* in this module description and the function description for *`reload/2`*.

The function throws a `badarg` exception if the parameters are not specified as described here.

```
try_load(Path, Name, OptionList) ->
            {ok, Status} |
            {ok, PendingStatus, Ref} |
            {error, ErrorDesc}
```

Types:

```
Path = path()
Name = driver()
OptionList = [Option]
Option =
    {driver_options, DriverOptionList} |
    {monitor, MonitorOption} |
    {reload, ReloadOption}
DriverOptionList = [DriverOption]
DriverOption = kill_ports
MonitorOption = ReloadOption = pending_driver | pending
Status = loaded | already_loaded | PendingStatus
PendingStatus = pending_driver | pending_process
Ref = reference()
ErrorDesc = ErrorAtom | OpaqueError
ErrorAtom =
    linked_in_driver |
    inconsistent |
    permanent |
    not_loaded_by_this_process |
    not_loaded |
    pending_reload |
```

pending_process
OpaqueError = term()

Provides more control than the `load/2`/`reload/2` and `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:

`{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.

```
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 trigging 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_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.

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

The Erlang **error logger** is an event manager (see *OTP Design Principles* and *gen_event(3)*), registered as `error_logger`. Errors, warnings, and info events are sent to the error logger from the Erlang runtime system and the different Erlang/OTP applications. The events are, by default, logged to the terminal. Notice that an event from a process `P` is logged at the node of the group leader of `P`. This means that log output is directed to the node from which a process was created, which not necessarily is the same node as where it is executing.

Initially, `error_logger` has only a primitive event handler, which buffers and prints the raw event messages. During system startup, the Kernel application replaces this with a **standard event handler**, by default one that writes nicely formatted output to the terminal. Kernel can also be configured so that events are logged to a file instead, or not logged at all, see *kernel(6)*.

Also the SASL application, if started, adds its own event handler, which by default writes supervisor, crash, and progress reports to the terminal. See *sasl(6)*.

It is recommended that user-defined applications report errors through the error logger to get uniform reports. User-defined event handlers can be added to handle application-specific events, see *add_report_handler/1,2*. Also, a useful event handler is provided in STDLIB for multi-file logging of events, see *log_mf_h(3)*.

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*.

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)`*.

error_msg(Format) -> ok
error_msg(Format, Data) -> ok
format(Format, Data) -> ok

Types:

Format = string()
Data = list()

Sends a standard error event to the error logger. The `Format` and `Data` arguments are the same as the arguments of *`io:format/2`* in STDLIB. The event is handled by the standard event handler.

Example:

```
1> error_logger:error_msg("An error occurred in ~p~n", [a_module]).

=ERROR REPORT==== 11-Aug-2005::14:03:19 ===
An error occurred in a_module
ok
```

Warning:

If called with bad arguments, this function can crash the standard event handler, meaning no further events are logged. When in doubt, use *`error_report/1`* instead.

Warning:

If the Unicode translation modifier (`t`) is used in the format string, all error handlers must ensure that the formatted output is correctly encoded for the I/O device.

error_report(Report) -> ok

Types:

Report = *report()*

Sends a standard error report event to the error logger. The event is handled by the standard event handler.

Example:

```
2> error_logger:error_report([{tag1,data1},a_term,{tag2,data}]).

=ERROR REPORT==== 11-Aug-2005::13:45:41 ===
    tag1: data1
    a_term
    tag2: data
ok
3> error_logger:error_report("Serious error in my module").

=ERROR REPORT==== 11-Aug-2005::13:45:49 ===
Serious error in my module
ok
```

error_report(Type, Report) -> ok

Types:

Type = term()
Report = *report()*

Sends a user-defined error report event to the error logger. An event handler to handle the event is supposed to have been added. The event is ignored by the standard event handler.

It is recommended that `Report` follows the same structure as for *`error_report/1`*.

get_format_depth() -> unlimited | integer() >= 1

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.

info_msg(Format) -> ok
info_msg(Format, Data) -> ok

Types:

Format = string()
Data = list()

Sends a standard information event to the error logger. The `Format` and `Data` arguments are the same as the arguments of *`io:format/2`* in STDLIB. The event is handled by the standard event handler.

Example:

```
1> error_logger:info_msg("Something happened in ~p~n", [a_module]).

=INFO REPORT==== 11-Aug-2005::14:06:15 ===
Something happened in a_module
ok
```

Warning:

If called with bad arguments, this function can crash the standard event handler, meaning no further events are logged. When in doubt, use `info_report/1` instead.

> **Warning:**
>
> If the Unicode translation modifier (`t`) is used in the format string, all error handlers must ensure that the formatted output is correctly encoded for the I/O device.

`info_report(Report) -> ok`

Types:

`Report = report()`

Sends a standard information report event to the error logger. The event is handled by the standard event handler.

Example:

```
2> error_logger:info_report([{tag1,data1},a_term,{tag2,data}]).

=INFO REPORT==== 11-Aug-2005::13:55:09 ===
    tag1: data1
    a_term
    tag2: data
ok
3> error_logger:info_report("Something strange happened").

=INFO REPORT==== 11-Aug-2005::13:55:36 ===
Something strange happened
ok
```

`info_report(Type, Report) -> ok`

Types:

`Type = any()`

`Report = report()`

Sends a user-defined information report event to the error logger. An event handler to handle the event is supposed to have been added. The event is ignored by the standard event handler.

It is recommended that `Report` follows the same structure as for *`info_report/1`*.

`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 standard event handler for output to file. Thus, calling this function overrides the value of the Kernel `error_logger` configuration parameter.

Enabling file logging can be used together with calling `tty(false)`, to have a silent system where all standard events are logged to a file only. Only one log file can be active at a time.

`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.

tty(Flag) -> ok

Types:

Flag = boolean()

Enables (`Flag == true`) or disables (`Flag == false`) printout of standard events to the terminal.

This is done by adding or deleting the standard event handler for output to the terminal. Thus, calling this function overrides the value of the Kernel `error_logger` configuration parameter.

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:

```
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()

Sends a standard warning event to the error logger. The `Format` and `Data` arguments are the same as the arguments of *`io:format/2`* in STDLIB. The event is handled by the standard event handler. It is tagged as an error, warning, or info, see *`warning_map/0`*.

Warning:

If called with bad arguments, this function can crash the standard event handler, meaning no further events are logged. When in doubt, use `warning_report/1` instead.

Warning:

If the Unicode translation modifier (`t`) is used in the format string, all error handlers must ensure that the formatted output is correctly encoded for the I/O device.

warning_report(Report) -> ok

Types:

Report = *report()*

Sends a standard warning report event to the error logger. The event is handled by the standard event handler. It is tagged as an error, warning, or info, see *`warning_map/0`*.

warning_report(Type, Report) -> ok

Types:

Type = any()

Report = *report()*

Sends a user-defined warning report event to the error logger. An event handler to handle the event is supposed to have been added. The event is ignored by the standard event handler. It is tagged as an error, warning, or info, depending on the value of *`warning_map/0`*.

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.

`{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)`, *`log_mf_h(3)`* *`kernel(6)`* *`sasl(6)`*

file

Erlang module

This module provides an interface to the file system.

On operating systems with thread support, file operations can be performed in threads of their own, allowing other Erlang processes to continue executing in parallel with the file operations. See command-line flag `+A` in *`erl(1)`*.

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 *`erts: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.

Data Types

`deep_list() = [char() | atom() | deep_list()]`

`fd()`

A file descriptor representing a file opened in *`raw`* mode.

`filename() = string()`

`filename_all() = string() | binary()`

`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.

```
name_all() =
    string() | atom() | deep_list() | (RawFilename :: binary())
```

If VM is in Unicode filename mode, `string()` and `char()` 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).

```
posix() =
    eacces |
```

```
    eagain |
    ebadf |
    ebusy |
    edquot |
    eexist |
    efault |
    efbig |
    eintr |
    einval |
    eio |
    eisdir |
    eloop |
    emfile |
    emlink |
    enametoolong |
    enfile |
    enodev |
    enoent |
    enomem |
    enospc |
    enotblk |
    enotdir |
    enotsup |
    enxio |
    eperm |
    epipe |
    erofs |
    espipe |
    esrch |
    estale |
    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,
                inode = 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. When it succeeds, space is preallocated for the file but the file size might not be updated. This behaviour depends on the preallocation implementation. To guarantee that the file size is updated, truncate the file to the new size.

`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`*.

`change_mode(Filename, Mode) -> ok | {error, Reason}`

Types:

`Filename = name_all()`
`Mode = integer()`
`Reason = posix() | badarg`

Changes permissions of a file. See *`write_file_info/2`*.

`change_owner(Filename, Uid) -> ok | {error, Reason}`

Types:

`Filename = name_all()`
`Uid = integer()`
`Reason = posix() | badarg`

Changes owner of a file. See *`write_file_info/2`*.

`change_owner(Filename, Uid, Gid) -> ok | {error, Reason}`

Types:

```
Filename = name_all()
Uid = Gid = integer()
Reason = posix() | badarg
```

Changes owner and group of a file. See *write_file_info/2*.

change_time(Filename, Mtime) -> ok | {error, Reason}

Types:

```
Filename = name_all()
Mtime = date_time()
Reason = posix() | badarg
```

Changes the modification and access times of a file. See *write_file_info/2*.

change_time(Filename, Atime, Mtime) -> ok | {error, Reason}

Types:

```
Filename = name_all()
Atime = Mtime = date_time()
Reason = posix() | badarg
```

Changes the modification and last access times 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.

`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.

`eacces`

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:

`eacces`

Missing read permission for one of the parents of the current directory.

`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`.

`eexist`

`New` already exists.

`enotsup`

Hard links are not supported on this platform.

`make_symlink(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]
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 *`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 *`unicode(3)`*.

This option is not allowed on `raw` files.

`ram`

`File` must be `iodata()`. Returns an `fd()`, which lets module `file` operate on the data in-memory as if it is a file.

`sync`

On platforms supporting it, enables the POSIX `O_SYNC` synchronous I/O flag or its platform-dependent equivalent (for example, `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 `sync` flag causes `open` to return `{error, enotsup}`.

Returns:

`{ok, IoDevice}`

The file is opened in the requested mode. `IoDevice` is a reference to the file.

`{error, Reason}`

The file cannot be opened.

`IoDevice` is really the pid of the process that handles the file. This process is linked to the process that originally opened the file. If any process to which the `IoDevice` is linked terminates, the file is closed and the process itself is terminated. An `IoDevice` returned from this call can be used as an argument to the I/O functions (see *`io(3)`*).

Note:

In previous versions of `file`, modes were specified as one of the atoms `read`, `write`, or `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 `read_write` is not allowed in a mode list.

Typical error reasons:

`enoent`

The file does not exist.

`eacces`

Missing permission for reading the file or searching one of the parent directories.

`eisdir`

The named file is not a regular file. It can be a directory, a FIFO, or a device.

`enotdir`

A component of the filename is not a directory. On some platforms, `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:

`{ok, FullName}`

The file is read and evaluated. `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 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_open(Path, Filename, Modes) ->
             {ok, IoDevice, FullName} | {error, Reason}
```

Types:

```
   Path = [Dir :: name_all()]
   Filename = name_all()
   Modes = [mode()]
   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.

```
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.

Warning:

This function is intended for debugging only.

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 `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 `pread`, is either a list or a binary depending on the mode of the file, or `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 `encoding` is set to something else than `latin1`, as not every byte position is a valid character boundary on such a file.

`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 `position/2` and `read/2` in one operation, which is more efficient than calling them one at a time. If `IoDevice` is opened in `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 `encoding` is set to something else than `latin1`, as not every byte position is a valid character boundary on such a file.

`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 `pwrite/3` in one operation, which is more efficient than calling them one at a time. Returns `ok` or `{error, {N, Reason}}`, where `N` is the number of successful writes done before the failure.

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`*.

pwrite(IoDevice, Location, Bytes) -> ok | {error, Reason}

Types:

IoDevice = *io_device()*
Location = *location()*
Bytes = iodata()
Reason = *posix()* | badarg | terminated

Combines `position/2` and `write/2` in one operation, which is more efficient than calling them one at a time. If `IoDevice` has been opened in `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(Filename) -> {ok, FileInfo} | {error, Reason}`
`read_file_info(Filename, Opts) -> {ok, FileInfo} | {error, Reason}`

Types:

`Filename = name_all()`
`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:

```
-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:

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:

`8#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.

`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 *`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 `raw` mode. However, it is inefficient to use it on `raw` files if the file is not opened with option `{read_ahead, Size}` specified. Thus, combining `raw` and `{read_ahead, Size}` is highly recommended when opening a text file for raw line-oriented reading.

If `encoding` is set to something else than `latin1`, the `read_line/1` call fails if the data contains characters larger than 255, why module *`io(3)`* is to be preferred when reading such a file.

The function returns:

`{ok, Data}`

One line from the file is returned, including the trailing LF, but with CRLF sequences replaced by a single LF (see above).

If the file is opened in binary mode, the read bytes are returned in a binary, otherwise in a list.

`eof`

Returned if 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 on the file cannot be translated to the byte-oriented data that this function returns.

`read_link(Name) -> {ok, Filename} | {error, Reason}`

Types:

`Name =` ***`name_all()`***
`Filename =` ***`filename()`***
`Reason =` ***`posix()`*** **`| badarg`**

Returns `{ok, Filename}` if `Name` refers to a symbolic link that is not a raw filename, or `{error, Reason}` otherwise. On platforms that do not support symbolic links, the return value is `{error,enotsup}`.

Typical error reasons:

`einval`

`Name` does not refer to a symbolic link or the name of the file that it refers to does not conform to the expected encoding.

`enoent`

The file does not exist.

`enotsup`

Symbolic links are not supported on this platform.

`read_link_all(Name) -> {ok, Filename} | {error, Reason}`

Types:

`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.

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 `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.

`use_threads`

Instructs the emulator to use the `async` thread pool for the `sendfile` system call. This can be useful if the OS you are running on does not properly support non-blocking `sendfile` calls. Notice that using `async` threads potentially makes your system vulnerable to slow client attacks. If set to `true` and no `async` threads are available, the `sendfile` call returns `{error,einval}`. Introduced in Erlang/OTP 17.0. Default is `false`.

`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:

enoent

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.

`write_file(Filename, Bytes, Modes) -> ok | {error, Reason}`

Types:

`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:

`8#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
- `estale` - Stale remote file handle
- `exdev` - Cross-domain link

Performance

Some operating system file operations, for example, a `sync/1` or `close/1` on a huge file, can block their calling thread for seconds. If this affects the emulator main thread, the response time is no longer in the order of milliseconds, depending on the definition of "soft" in soft real-time system.

If the device driver thread pool is active, file operations are done through those threads instead, so the emulator can go on executing Erlang processes. Unfortunately, the time for serving a file operation increases because of the extra scheduling required from the operating system.

If the device driver thread pool is disabled or of size 0, large file reads and writes are segmented into many smaller, which enable the emulator to serve other processes during the file operation. This has the same effect as when using the thread pool, but with larger overhead. Other file operations, for example, `sync/1` or `close/1` on a huge file, still are a problem.

For increased performance, raw files are recommended. Raw files use the file system of the host machine of the node.

Note:

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.

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.

Caching data to reduce the number of file operations, or rather the number of calls to the file driver, generally increases performance. The following function writes 4 MBytes in 23 seconds when tested:

```
create_file_slow(Name, N) when integer(N), N >= 0 ->
    {ok, FD} = file:open(Name, [raw, write, delayed_write, binary]),
    ok = create_file_slow(FD, 0, N),
    ok = ?FILE_MODULE:close(FD),
    ok.

create_file_slow(FD, M, M) ->
    ok;
create_file_slow(FD, M, N) ->
    ok = file:write(FD, <<M:32/unsigned>>),
    create_file_slow(FD, M+1, N).
```

The following, functionally equivalent, function collects 1024 entries into a list of 128 32-byte binaries before each call to *`write/2`* and so does the same work in 0.52 seconds, which is 44 times faster:

```
create_file(Name, N) when integer(N), N >= 0 ->
    {ok, FD} = file:open(Name, [raw, write, delayed_write, binary]),
    ok = create_file(FD, 0, N),
    ok = ?FILE_MODULE:close(FD),
    ok.

create_file(FD, M, M) ->
    ok;
create_file(FD, M, N) when M + 1024 =&lt; N ->
    create_file(FD, M, M + 1024, []),
    create_file(FD, M + 1024, N);
create_file(FD, M, N) ->
    create_file(FD, M, N, []).

create_file(FD, M, M, R) ->
    ok = file:write(FD, R);
create_file(FD, M, N0, R) when M + 8 =&lt; N0 ->
    N1  = N0-1,  N2  = N0-2,  N3  = N0-3,  N4  = N0-4,
    N5  = N0-5,  N6  = N0-6,  N7  = N0-7,  N8  = N0-8,
    create_file(FD, M, N8,
                [<<N8:32/unsigned,  N7:32/unsigned,
                   N6:32/unsigned,  N5:32/unsigned,
                   N4:32/unsigned,  N3:32/unsigned,
                   N2:32/unsigned,  N1:32/unsigned>> | R]);
create_file(FD, M, N0, R) ->
    N1 = N0-1,
    create_file(FD, M, N1, [<<N1:32/unsigned>> | R]).
```

Note:

Trust only your own benchmarks. If the list length in `create_file/2` above is increased, it runs slightly faster, but consumes more memory and causes more memory fragmentation. How much this affects your application is something that this simple benchmark cannot predict.

If the size of each binary is increased to 64 bytes, it also runs slightly faster, but the code is then twice as clumsy. In the current implementation, binaries larger than 64 bytes are stored in memory common to all processes and not copied when sent between processes, while these smaller binaries are stored on the process heap and copied when sent like any other term.

So, with a binary size of 68 bytes, `create_file/2` runs 30 percent slower than with 64 bytes, and causes much more memory fragmentation. Notice that if the binaries were to be sent between processes (for example, a non-raw file), the results would probably be completely different.

A raw file is really a port. When writing data to a port, it is efficient to write a list of binaries. 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.

For raw files, `pwrite/2` and `pread/2` are efficiently implemented. The file driver is called only once for the whole operation, and the list iteration is done in the file driver.

The options `delayed_write` and `read_ahead` to *`open/2`* make the file driver cache data to reduce the number of operating system calls. The function `create_file/2` in the recent example takes 60 seconds without option `delayed_write`, which is 2.6 times slower.

As a bad example, `create_file_slow/2` without options `raw`, `binary`, and `delayed_write`, meaning it calls `open(Name, [write])`, needs 1 min 20 seconds for the job, which is 3.5 times slower than the first example, and 150 times slower than the optimized `create_file/2`.

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)

gen_sctp

Erlang module

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:

```
-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".

```
option() =
    {active, true | false | once | -32768..32767} |
    {buffer, integer() >= 0} |
    {dontroute, 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{}} |
    {sctp_disable_fragments, boolean()} |
    {sctp_events, #sctp_event_subscribe{}} |
    {sctp_get_peer_addr_info, #sctp_paddrinfo{}} |
    {sctp_i_want_mapped_v4_addr, boolean()} |
    {sctp_initmsg, #sctp_initmsg{}} |
    {sctp_maxseg, integer() >= 0} |
    {sctp_nodelay, boolean()} |
    {sctp_peer_addr_params, #sctp_paddrparams{}} |
    {sctp_primary_addr, #sctp_prim{}} |
    {sctp_rtoinfo, #sctp_rtoinfo{}} |
    {sctp_set_peer_primary_addr, #sctp_setpeerprim{}} |
    {sctp_status, #sctp_status{}} |
    {sndbuf, integer() >= 0} |
    {tos, integer() >= 0}
```

One of the *SCTP Socket Options*.

```
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_status |
    sndbuf |
    tos
```

```
sctp_socket()
```

Socket identifier returned from *`open/*`*.

Exports

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.

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.

connect(Socket, Addr, Port, Opts) ->
{ok, Assoc} | {error, *inet:posix()*}

Types:

Socket = *sctp_socket()*
Addr = *inet:ip_address()* | *inet:hostname()*
Port = *inet:port_number()*
Opts = [Opt :: *option()*]
Assoc = #sctp_assoc_change{}

Same as `connect(Socket, Addr, Port, Opts, infinity)`.

connect(Socket, Addr, Port, Opts, Timeout) ->
{ok, Assoc} | {error, *inet:posix()*}

Types:

Socket = *sctp_socket()*
Addr = *inet:ip_address()* | *inet:hostname()*
Port = *inet:port_number()*
Opts = [Opt :: *option()*]
Timeout = timeout()
Assoc = #sctp_assoc_change{}

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*:

```
#sctp_assoc_change{
      state             = atom(),
      error             = atom(),
      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:

```
connect(Socket, Ip, Port>,
      [{sctp_initmsg,#sctp_initmsg{num_ostreams=OutStreams,
                                   max_instreams=MaxInStreams}}])
```

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.

`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 `{ip,IP}` (or synonymously `{ifaddr,IP}`) options (this feature is called SCTP multi-homing). The default `IP` and `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 `seqpacket`, and with reasonably large *kernel* and driver *buffers*.

```
peeloff(Socket, Assoc) -> {ok, NewSocket} | {error, Reason}
```

Types:

```
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{}]
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).

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{}`*

- ```
  #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`*. 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.
  ```

- ```
  #sctp_pdapi_event{
        indication = sctp_partial_delivery_aborted,
        assoc_id   = assoc_id()
  }
  ```

  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` (full active mode), the pending data or events are sent to the owning process.

  Notice that this can cause the message queue to overflow, as there is no way to throttle the sender in this case (no flow control).
- 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.

`{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.

`{priority, integer()}`

A protocol-independent equivalent of `tos` above. Setting priority implies setting `tos` as well.

`{dontroute, true|false}`

Defaults to `false`. If `true`, the kernel does not send packets through any gateway, only sends them to directly connected hosts.

`{reuseaddr, true|false}`

Defaults to `false`. If true, the local binding address `{IP,Port}` of the socket can be reused immediately. No waiting in state `CLOSE_WAIT` is performed (can be required for high-throughput servers).

`{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).

`{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).

`{sctp_module, module()}`

Overrides which callback module is used. Defaults to `inet_sctp` for IPv4 and `inet6_sctp` for IPv6.

`{sctp_rtoinfo, #sctp_rtoinfo{}}`

```
#sctp_rtoinfo{
      assoc_id = assoc_id(),
      initial  = integer(),
      max      = integer(),
      min      = integer()
}
```

Determines retransmission time-out parameters, in milliseconds, for the association(s) specified by `assoc_id`.

`assoc_id = 0` (default) indicates the whole endpoint. See **RFC 2960** and **Sockets API Extensions for SCTP** for the exact semantics of the field values.

`{sctp_associnfo, #sctp_assocparams{}}`

```
#sctp_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{}}`

```
#sctp_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. 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{}}`

```
#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{}}`

```
#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{}}`

```
#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 Extenstions for SCTP**.

`{sctp_peer_addr_params, #sctp_paddrparams{}}`

```
#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()
}
```

`#sctp_sndrcvinfo{}` is used both in this socket option, and as ancillary data while sending or receiving SCTP messages. When set as an option, it provides default values for subsequent *`send`* calls on the association specified by `assoc_id`.

`assoc_id = 0` (default) indicates the whole endpoint.

The following fields typically must be specified by the sender:

`sinfo_stream`

Stream number (0-base) within the association to send the messages through;

`sinfo_flags`

The following flags are recognised:

`unordered`
: The message is to be sent unordered

`addr_over`
: The address specified in *`send`* overwrites the primary peer address

`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` fmust 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:

```
-module(sctp_server).

-export([server/0,server/1,server/2]).
-include_lib("kernel/include/inet.hrl").
-include_lib("kernel/include/inet_sctp.hrl").

server() ->
    server(any, 2006).

server([Host,Port]) when is_list(Host), is_list(Port) ->
    {ok, #hostent{h_addr_list = [IP|_]}} = inet:gethostbyname(Host),
    io:format("~w -> ~w~n", [Host, IP]),
    server([IP, list_to_integer(Port)]).

server(IP, Port) when is_tuple(IP) orelse IP == any orelse IP == loopback,
                      is_integer(Port) ->
    {ok,S} = gen_sctp:open(Port, [{recbuf,65536}, {ip,IP}]),
    io:format("Listening on ~w:~w. ~w~n", [IP,Port,S]),
    ok     = gen_sctp:listen(S, true),
    server_loop(S).

server_loop(S) ->
    case gen_sctp:recv(S) of
    {error, Error} ->
        io:format("SCTP RECV ERROR: ~p~n", [Error]);
    Data ->
        io:format("Received: ~p~n", [Data])
    end,
    server_loop(S).
```

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 `abort`s the association, which results in that the corresponding event is received on the server side.

```
-module(sctp_client).

-export([client/0, client/1, client/2]).
-include_lib("kernel/include/inet.hrl").
-include_lib("kernel/include/inet_sctp.hrl").

client() ->
    client([localhost]).

client([Host]) ->
    client(Host, 2006);

client([Host, Port]) when is_list(Host), is_list(Port) ->
    client(Host,list_to_integer(Port)),
    init:stop().

client(Host, Port) when is_integer(Port) ->
    {ok,S}     = gen_sctp:open(),
    {ok,Assoc} = gen_sctp:connect
        (S, Host, Port, [{sctp_initmsg,#sctp_initmsg{num_ostreams=5}}]),
    io:format("Connection Successful, Assoc=~p~n", [Assoc]),

    io:write(gen_sctp:send(S, Assoc, 0, <<"Test 0">>)),
    io:nl(),
    timer:sleep(10000),
    io:write(gen_sctp:send(S, Assoc, 5, <<"Test 5">>)),
    io:nl(),
    timer:sleep(10000),
    io:write(gen_sctp:abort(S, Assoc)),
    io:nl(),

    timer:sleep(1000),
    gen_sctp:close(S).
```

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**

gen_tcp

Erlang module

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:

```
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:

```
server() ->
    {ok, LSock} = gen_tcp:listen(5678, [binary, {packet, 0},
                                        {active, false}]),
    {ok, Sock} = gen_tcp:accept(LSock),
    {ok, 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

```
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() >= 1} |
    {low_watermark, integer() >= 0} |
    {mode, list | binary} |
    list |
    binary |
    {nodelay, boolean()} |
```

```
    {packet,
     0 |
     1 |
     2 |
     4 |
     raw |
     sunrm |
     asn1 |
     cdr |
     fcgi |
     line |
     tpkt |
     http |
     httph |
     http_bin |
     httph_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} |
    {ipv6_v6only, boolean()}
option_name() =
    active |
    buffer |
    delay_send |
    deliver |
    dontroute |
    exit_on_close |
    header |
    high_msgq_watermark |
    high_watermark |
    keepalive |
    linger |
    low_msgq_watermark |
    low_watermark |
    mode |
    nodelay |
    packet |
    packet_size |
    priority |
    {raw,
     Protocol :: integer() >= 0,
     OptionNum :: integer() >= 0,
```

```
     ValueSpec ::
         (ValueSize :: integer() >= 0) | (ValueBin :: binary())} |
    recbuf |
    reuseaddr |
    send_timeout |
    send_timeout_close |
    show_econnreset |
    sndbuf |
    tos |
    ipv6_v6only
connect_option() =
    {ip, inet:socket_address()} |
    {fd, Fd :: integer() >= 0} |
    {ifaddr, inet:socket_address()} |
    inet:address_family() |
    {port, inet:port_number()} |
    {tcp_module, module()} |
    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()} |
    option()
socket()
```

As returned by *accept/1,2* and *connect/3,4*.

Exports

```
accept(ListenSocket) -> {ok, Socket} | {error, Reason}
accept(ListenSocket, Timeout) -> {ok, Socket} | {error, Reason}
```

Types:

ListenSocket = *socket()*

Returned by *listen/2*.

Timeout = timeout()

Socket = *socket()*

Reason = closed | timeout | system_limit | *inet:posix()*

Accepts an incoming connection request on a listening socket. `Socket` must be a socket returned from *listen/2*. `Timeout` specifies a time-out value in milliseconds. Defaults to `infinity`.

Returns:

- `{ok, Socket}` if a connection is established
- `{error, closed}` if `ListenSocket` is closed
- `{error, timeout}` if no connection is established within the specified time
- `{error, system_limit}` if all available ports in the Erlang emulator are in use

- A POSIX error value if something else goes wrong, see *inet(3)* for possible error values

Packets can be sent to the returned socket `Socket` using *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 *recv/2*):

```
{tcp, Socket, Data}
```

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(Sock, 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 knowns 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 = 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:

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 transfere is complete may cause this.

listen(Port, Options) -> {ok, ListenSocket} | {error, Reason}

Types:

Port = *inet:port_number()*
Options = [*listen_option()*]
ListenSocket = *socket()*
Reason = system_limit | *inet:posix()*

Sets up a socket to listen on port `Port` on the local host.

If `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.

`{backlog, B}`

`B` is an integer >= `0`. The backlog value defines the maximum length that the queue of pending connections can grow to. Defaults to `5`.

`{ip, Address}`

If the host has many network interfaces, this option specifies which one to listen on.

`{port, Port}`

Specifies which local port number to use.

`{fd, Fd}`

If a socket has somehow been connected without using `gen_tcp`, use this option to pass the file descriptor for it.

`{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.

`{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`*.

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() >= 0`
`Timeout = timeout()`
`Packet = string() | binary() | HttpPacket`
`Reason = closed | 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 *`recv/2`* or *`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 `start/2` takes the number of worker processes and the port number on which to listen for incoming connections. If `LPort` is specified as `0`, an ephemeral port number is used, which is why the start function returns the actual port number allocated:

```
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.
```

Example of a simple client:

```
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.
```

The `send` call does not accept a time-out option because time-outs on send is handled through socket option `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:

```
...
{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:

```
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 acknowledgment from the server, but if the protocol is strictly one way, option `send_timeout` comes in handy.

gen_udp

Erlang module

This module provides functions for communicating with sockets using the UDP protocol.

Data Types

```
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} |
    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} |
    {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 |
    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()} |
    option()
Socket = socket()
Reason = 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}
```

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 *inet:setopts/2* for details), transitions to passive (`{active, false}`) mode, the controlling process is notified by a message of the following form:

```
{udp_passive, Socket}
```

`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 *inet:port/1* to retrieve it.

recv(Socket, Length) ->
{ok, {Address, Port, Packet}} | {error, Reason}
recv(Socket, Length, Timeout) ->
{ok, {Address, Port, Packet}} | {error, Reason}

Types:

Socket = *socket()*
Length = integer() >= 0
Timeout = timeout()
Address = *inet:ip_address()* | *inet:returned_non_ip_address()*
Port = *inet:port_number()*
Packet = string() | binary()
Reason = not_owner | *inet:posix()*

Receives a packet from a socket in passive mode. Optional parameter `Timeout` specifies a time-out in milliseconds. Defaults to `infinity`.

send(Socket, Address, Port, Packet) -> ok | {error, Reason}

Types:

Socket = *socket()*
Address = *inet:socket_address()* | *inet:hostname()*
Port = *inet:port_number()*
Packet = iodata()
Reason = not_owner | *inet:posix()*

Sends a packet to the specified address and port. Argument `Address` can be a hostname or a socket address.

global

Erlang module

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() = {ResourceId :: term(), LockRequesterId :: term()}

Exports

del_lock(Id) -> true
del_lock(Id, Nodes) -> true

Types:

Id = *id()*
Nodes = [node()]

Deletes the lock `Id` synchronously.

notify_all_name(Name, Pid1, Pid2) -> none

Types:

Name = term()
Pid1 = Pid2 = pid()

Can be used as a name resolving function for *`register_name/3`* and *`re_register_name/3`*.

The function unregisters both pids and sends the message `{global_name_conflict, Name, OtherPid}` to both processes.

random_exit_name(Name, Pid1, Pid2) -> pid()

Types:

Name = term()
Pid1 = Pid2 = pid()

Can be used as a name resolving function for *`register_name/3`* and *`re_register_name/3`*.

The function randomly selects one of the pids for registration and kills the other one.

random_notify_name(Name, Pid1, Pid2) -> pid()

Types:

Name = term()
Pid1 = Pid2 = pid()

Can be used as a name resolving function for *`register_name/3`* and *`re_register_name/3`*.

The function randomly selects one of the pids for registration, and sends the message `{global_name_conflict, Name}` to the other pid.

re_register_name(Name, Pid) -> yes
re_register_name(Name, Pid, Resolve) -> yes

Types:

Name = term()
Pid = pid()
Resolve = *method()*
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_log_writes`
- `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)`*

global_group

Erlang module

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()]} |
    {synched_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, Node}` and `{nodedown, Node}` when a group node connects or disconnects, respectively.

`own_nodes() -> Nodes`

Types:

`Nodes = [Node :: node()]`

Returns the names of all group nodes, regardless of their current status.

`registered_names(Where) -> Names`

Types:

`Where = where()`

`Names = [Name :: name()]`

Returns a list of all names that are globally registered on the specified node or in the specified global group.

`send(Name, Msg) -> pid() | {badarg, {Name, Msg}}`

`send(Where, Name, Msg) -> pid() | {badarg, {Name, Msg}}`

Types:

`Where = where()`

`Name = name()`

`Msg = term()`

Searches for `Name`, globally registered on the specified node or in the specified global group, or (if argument `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 `global_groups`.

If `Name` is found, message `Msg` is sent to the corresponding pid. The pid is also the return value of the function. If the name is not found, the function returns `{badarg, {Name, Msg}}`.

`sync() -> 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 *`error_logger(3)`*.

Returns `{error, {'invalid global_groups definition', Bad}}` if configuration parameter `global_groups` has an invalid value `Bad`.

`whereis_name(Name) -> pid() | undefined`

`whereis_name(Where, Name) -> pid() | undefined`

Types:

`Where = where()`

`Name = name()`

Searches for `Name`, globally registered on the specified node or in the specified global group, or (if argument `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 `global_groups`.

If `Name` is found, the corresponding pid is returned. If the name is not found, the function returns `undefined`.

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 modules 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.

Notice that if the system clock is adjusted with more than `HEART_BEAT_TIMEOUT` seconds, `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 completly 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.

inet

Erlang module

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 '[{delay_send,true}]' \
inet_default_listen_options '[{delay_send,true}]'
```

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:

```
Address          ip_address()
-------          ------------
127.0.0.1        {127,0,0,1}
192.168.42.2     {192,168,42,2}
```

IPv6 address examples:

```
Address          ip_address()
-------          ------------
::1              {0,0,0,0,0,0,0,1}
::192.168.42.2   {0,0,0,0,0,0,(192 bsl 8) bor 168,(42 bsl 8) bor 2}
::FFFF:192.168.42.2
                 {0,0,0,0,0,16#FFFF,(192 bsl 8) bor 168,(42 bsl 8) bor 2}
3ffe:b80:1f8d:2:204:acff:fe17:bf38
                 {16#3ffe,16#b80,16#1f8d,16#2,16#204,16#acff,16#fe17,16#bf38}
fe80::204:acff:fe17:bf38
                 {16#fe80,0,0,0,0,16#204,16#acff,16#fe17,16#bf38}
```

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,{0,0,0,0,0,65535,49320,10754}}
```

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
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.

`posix() = exbadport | exbadseq | file:posix()`

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*.

`socket()`

See `gen_tcp:type-socket` and `gen_udp:type-socket`.

`address_family() = inet | inet6 | local`

Exports

`close(Socket) -> ok`

Types:

`Socket = socket()`

Closes a socket of any type.

`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. If that fails, the IPv4 address is looked up and returned on IPv6-mapped IPv4 format.

```
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, Iflist} | {error, posix()}
```

Types:

```
Iflist = [{Ifname, [Ifopt]}]
Ifname = string()
Ifopt =
    {flags, [Flag]} |
    {addr, Addr} |
    {netmask, Netmask} |
    {broadaddr, Broadaddr} |
    {dstaddr, Dstaddr} |
    {hwaddr, Hwaddr}
Flag =
```

```
      up | broadcast | loopback | pointtopoint | running | multicast
   Addr = Netmask = Broadaddr = Dstaddr = ip_address()
   Hwaddr = [byte()]
```

Returns a list of 2-tuples containing interface names and the interface addresses. `Ifname` is a Unicode string. `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,_}`, and `{broadaddr,_}` are repeated in the result list if the interface has multiple addresses. If you come across an interface with multiple `{flag,_}` or `{hwaddr,_}` tuples, you have a strange interface or possibly a bug in this function. The tuple `{flag,_}` is mandatory, all others are optional.

Do not rely too much on the order of `Flag` atoms or `Ifopt` tuples. There are however some rules:

- Immediately after `{addr,_}` follows `{netmask,_}`.
- Immediately thereafter follows `{broadaddr,_}` if flag `broadcast` is **not** set and flag `pointtopoint` **is** set.
- Any `{netmask,_}`, `{broadaddr,_}`, or `{dstaddr,_}` tuples that follow an `{addr,_}` tuple concerns that address.

The tuple `{hwaddr,_}` is not returned on Solaris, as the hardware address historically belongs to the link layer and only the superuser can read such addresses.

Warning:

On Windows, the data is fetched from different OS API functions, so the `Netmask` and `Broadaddr` values can be calculated, just as some `Flag` values. Report flagrant bugs.

```
getopts(Socket, Options) -> {ok, OptionValues} | {error, posix()}
```

Types:

```
   Socket = socket()
   Options = [socket_getopt()]
   OptionValues = [socket_setopt()]
```

Gets one or more options for a socket. For a list of available options, see *`setopts/2`*.

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 the current 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 endianess.

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 `tcpi_sacked` is 28 bytes, and the value is a 32-bit integer. The following code can be used to retrieve the value:

```
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.

`getstat(Socket) -> {ok, OptionValues} | {error, posix()}`
`getstat(Socket, Options) -> {ok, OptionValues} | {error, posix()}`

Types:

`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_dvi, 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_dvi`

Average packet size deviation, in bytes, sent from the socket.

`send_max`

Size of the largest packet, in bytes, sent from the socket.

`send_oct`

Number of bytes sent from the socket.

`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.

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 software buffer used by the driver. Not to be confused with options `sndbuf` and `recbuf`, which correspond to the Kernel socket buffers. It is recommended to have `val(buffer) >= max(val(sndbuf),val(recbuf))` to avoid performance issues because of unnecessary copying. `val(buffer)` is automatically set to the above maximum when values `sndbuf` or `recbuf` are set. However, as the sizes set for `sndbuf` and `recbuf` usually become larger, you are encouraged to use *`getopts/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 `port` : `{S, {data, [H1,..Hsz | Data]}}` or `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 *`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_msqg_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. If the first component of the value tuple is `false`, the second is ignored. This means that `close/1` returns immediately, not waiting for data to be flushed. Otherwise, the second component is the flushing time-out, in seconds.

`{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`*.

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.
- *`getopts/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**.

`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.

`{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}`.

`httph | httph_bin`

These two types are often not needed, as the socket automatically switches from `http`/`http_bin` to `httph`/`httph_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 *`getopts/2`* to retrieve the size set by your operating system.

`{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_close, 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 *`getopts/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()} |
               returned_non_ip_address()]} |
             {error, posix()}
```

Equivalent to *socknames(Socket, 0)*.

```
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 interprets 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
- `ebusy` - 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
- `eisdir` - 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
- `elnrng` - Link number out of range

- `eloop` - 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
- `enetdown` - 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
- `enosym` - Unresolved symbol name
- `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
- `eprototype` - 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

inet_res

Erlang module

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 hiearchy 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

query_type() = axfr | mailb | maila | any | rr_type()

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. If that fails, the IPv4 address is looked up and returned on IPv6-mapped IPv4 format.

```
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:

```
Name = dns_name() | inet:ip_address()
Class = dns_class()
Type = rr_type()
Opts = [Opt]
Opt = res_option() | verbose | atom()
Timeout = timeout()
Error = {error, Reason} | {error, {Reason, dns_msg()}}
Reason = inet:posix() | res_error()
```

Resolves a DNS record of the specified type and class for the specified name. The returned `dns_msg()` can be examined using access functions in `inet_db`, as described in section in *DNS Types*.

If `Name` is an `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 `Class = in` and `Type = ptr`, but it is not done automatically.

`Opts` overrides the corresponding resolver options. If option `nameservers` is specified, it is assumed that it is the complete list of name serves, so resolver option `alt_nameserves` is ignored. However, if option `alt_nameserves` is also specified to this function, it is used.

Option `verbose` (or rather `{verbose,true}`) causes diagnostics printout through *`io:format/2`* of queries, replies retransmissions, and so on, similar to from utilities, such as `dig` and `nslookup`.

If `Opt` is any atom, it is interpreted as `{Opt,true}` unless the atom string starts with `"no"`, making the interpretation `{Opt,false}`. For example, `usevc` is an alias for `{usevc,true}` and `nousevc` is an alias for `{usevc,false}`.

Option `inet6` has no effect on this function. You probably want to use `Type = a | aaaa` instead.

Example

This access functions example shows how *`lookup/3`* can be implemented using *`resolve/3`* from outside the module:

```
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.
```

Legacy Functions

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

```
nslookup(Name, Class, Type) -> {ok, dns_msg()} | {error, Reason}
nslookup(Name, Class, Type, Timeout) ->
            {ok, dns_msg()} | {error, Reason}
nslookup(Name, Class, Type, Nameservers) ->
```

```
            {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() | 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.

net_adm

Erlang module

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:

```
(arne@dunn)1> net_adm:names().
{ok,[{"arne",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)
```

net_kernel

Erlang module

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`:

```
$ erl -sname foobar
```

It is also possible to call `net_kernel:start([foobar])` directly from the normal Erlang shell prompt:

```
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 `false`, 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 successful, `false` if not, 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(Sock, Options)* where `Sock` 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 `OptionList == []`, 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 be 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.

```
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`)

`minimum(NetTicktime, PreviousNetTicktime)*1000 div 4` milliseconds.

Transition period

The time of the least number of consecutive `MTTI`s to cover `TransitionPeriod` seconds following the call to `set_net_ticktime/2` (that is, ((`TransitionPeriod*1000 - 1) div MTTI + 1)*MTTI` milliseconds).

If `NetTicktime < PreviousNetTicktime`, the `net_ticktime` change is done at the end of the transition period; otherwise at the beginning. During the transition period, `net_kernel` ensures that there is outgoing traffic on all connections at least every `MTTI` millisecond.

> **Note:**
>
> The `net_ticktime` changes must be initiated on all nodes in the network (with the same `NetTicktime`) before the end of any transition period on any node; otherwise connections can erroneously be disconnected.

Returns one of the following:

`unchanged`

`net_ticktime` already has the value of `NetTicktime` and is left unchanged.

`change_initiated`

`net_kernel` initiated the change of `net_ticktime` to `NetTicktime` seconds.

`{ongoing_change_to, NewNetTicktime}`

The request is **ignored** because `net_kernel` is busy changing `net_ticktime` to `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 configration 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 `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.

os

Erlang module

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.

Data Types

os_command() = atom() | *io_lib:chars()*

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.

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() -> [string()]

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 = Value = string()

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 = DefaultValue = Value = string()

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 = Value = string()

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.

`set_signal(Signal, Option) -> ok`

Types:

```
Signal =
    sighup |
    sigquit |
    sigabrt |
    sigalrm |
    sigterm |
    sigusr1 |
    sigusr2 |
    sigchld |
    sigstop |
    sigtstp
Option = default | handle | ignore
```

Enables or disables OS signals.

Each signal my 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*.

> **Note:**
>
> This time is **not** a monotonically increasing time.

`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)`.

> **Note:**
>
> This time is **not** a monotonically increasing time.

`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:

```
-module(print_time).
-export([format_utc_timestamp/0]).
format_utc_timestamp() ->
    TS = {_,_,Micro} = os:timestamp(),
    {{Year,Month,Day},{Hour,Minute,Second}} =
calendar:now_to_universal_time(TS),
    Mstr = element(Month,{"Jan","Feb","Mar","Apr","May","Jun","Jul",
    "Aug","Sep","Oct","Nov","Dec"}),
    io_lib:format("~2w ~s ~4w ~2w:~2..0w:~2..0w.~6..0w",
    [Day,Mstr,Year,Hour,Minute,Second,Micro]).
```

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 retreived 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
Osname = 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 = string()
```

Deletes the environment variable `VarName`.

If Unicode filename encoding is in effect (see the *`erl` manual page*), the string `VarName` can contain characters with codepoints > 255.

`version() -> VersionString | {Major, Minor, Release}`

Types:

```
VersionString = string()
Major = Minor = Release = integer() >= 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.

> **Note:**
>
> Think twice before using this function. If you still need to use it, always `call os:type()` first.

pg2

Erlang module

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)`

rpc

Erlang module

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.

Data Types

key()

As 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:

`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.

block_call(Node, Module, Function, Args) -> Res | {badrpc, Reason}

Types:

Node = node()
Module = module()
Function = atom()
Args = [term()]
Res = Reason = term()

Same as *call/4*, but the RPC server at Node does not create a separate process to handle the call. Thus, this function can be used if the intention of the call is to block the RPC server from any other incoming requests until the request has been handled. The function can also be used for efficiency reasons when very small fast functions are evaluated, for example, BIFs that are guaranteed not to suspend.

block_call(Node, Module, Function, Args, Timeout) ->
Res | {badrpc, Reason}

Types:

Node = node()
Module = module()
Function = atom()
Args = [term()]
Res = Reason = term()
Timeout = timeout()

Same as *block_call/4*, but with a time-out value in the same manner as *call/5*.

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.

call(Node, Module, Function, Args, Timeout) ->
Res | {badrpc, Reason}

Types:

```
Node = node()
Module = module()
Function = atom()
Args = [term()]
Res = Reason = term()
Timeout = timeout()
```

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, as this function spawns off a middleman process to act as (a void) destination for such an orphan reply. This feature also makes this function more expensive than `call/4` at the caller's end.

`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`*.

`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:

```
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 terminatd before sending any reply.

multicall(Module, Function, Args) -> {ResL, BadNodes}

Types:

Module = module()
Function = atom()
Args = ResL = [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 = ResL = [term()]
BadNodes = [node()]

Equivalent to `multicall(Nodes, Module, Function, Args, infinity)`.

multicall(Module, Function, Args, Timeout) -> {ResL, BadNodes}

Types:

Module = module()
Function = atom()
Args = [term()]
Timeout = timeout()
ResL = [term()]
BadNodes = [node()]

Equivalent to `multicall([node()|nodes()], Module, Function, Args, Timeout)`.

multicall(Nodes, Module, Function, Args, Timeout) ->
{ResL, BadNodes}

Types:

Nodes = [node()]
Module = module()
Function = atom()
Args = [term()]
Timeout = timeout()
ResL = [term()]
BadNodes = [node()]

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 terminated or timed out during computation, and `ResL` is a list of the return values. `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:

```
%% 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.
```

nb_yield(Key) -> {value, Val} | timeout

Types:

Key = *key()*
Val = (Res :: term()) | {badrpc, Reason :: term()}

Equivalent to `nb_yield(Key, 0)`.

nb_yield(Key, Timeout) -> {value, Val} | timeout

Types:

Key = *key()*
Timeout = timeout()
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.

> **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`.

pinfo(Pid) -> [{Item, Info}] | undefined

Types:

Pid = pid()
Item = atom()
Info = term()

Location transparent version of the BIF *erlang:process_info/1* in ERTS.

pinfo(Pid, Item) -> {Item, Info} | undefined | []
pinfo(Pid, ItemList) -> [{Item, Info}] | undefined | []

Types:

Pid = pid()
Item = atom()
ItemList = [Item]
Info = term()

Location transparent version of the BIF *erlang:process_info/2* in ERTS.

pmap(FuncSpec, ExtraArgs, List1) -> List2

Types:

```
FuncSpec = {Module, Function}
Module = module()
Function = atom()
ExtraArgs = [term()]
List1 = [Elem :: term()]
List2 = [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`.

`sbcast(Name, Msg) -> {GoodNodes, BadNodes}`

Types:

```
Name = atom()
Msg = term()
GoodNodes = BadNodes = [node()]
```

Equivalent to `sbcast([node()|nodes()], Name, Msg)`.

`sbcast(Nodes, Name, Msg) -> {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.

seq_trace

Erlang module

Sequential tracing makes it possible to trace all messages resulting from one initial message. Sequential tracing is independent of the ordinary tracing in Erlang, which is controlled by the `erlang:trace/3` BIF. For more information about what sequential tracing is and how it can be used, see section *Sequential Tracing*.

`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

`token() = {integer(), boolean(), term(), term(), term()}`

An opaque term (a tuple) representing a trace token.

Exports

`set_token(Token) -> PreviousToken | ok`

Types:

`Token = PreviousToken = [] | token()`

Sets the trace token for the calling process to `Token`. If `Token == []` then tracing is disabled, otherwise `Token` should be an Erlang term returned from `get_token/0` or `set_token/1`. `set_token/1` can be used to temporarily exclude message passing from the trace by setting the trace token to empty like this:

```
OldToken = seq_trace:set_token([]), % set to empty and save
                                    % old value
% do something that should not be part of the trace
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.

`set_token(Component, Val) -> {Component, OldVal}`

Types:

`Component = component()`

`Val = OldVal = value()`

`component() = label | serial | flag()`

`flag() =`
`    send |`
`    'receive' |`
`    print |`
`    timestamp |`
`    monotonic_timestamp |`
`    strict_monotonic_timestamp`

`value() =`
`    (Integer :: integer() >= 0) |`
`    {Previous :: integer() >= 0, Current :: integer() >= 0} |`

(Bool :: boolean())

Sets the individual `Component` of the trace token to `Val`. Returns the previous value of the component.

`set_token(label, Integer)`

The `label` component is an integer 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.

`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 message sending. Default is `false`.

`set_token('receive', Bool)`

A trace token flag (`true | false`) which enables/disables tracing on message 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() =
    (Integer :: integer() >= 0) |
    {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 `print` set to `true` has sent a message.

`{'receive', Serial, From, To, Message}`

Used when a process `To` receives a message with a trace token that has flag `'receive'` set to `true`.

`{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 message 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 messages sent between different local or remote processes, where the sequence is initiated by a single message. 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 `{Label, Flags, Serial, From}`. The trace token is passed invisibly with each message.

To start a sequential trace, the user must explicitly set the trace token in the process that will send the first message in a sequence.

The trace token of a process is set each time 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 a message 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.

Trace Token

Each process has a current trace token. Initially, the token is empty. When the process sends a message to another process, a copy of the current token is sent "invisibly" along with the message.

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 a message is received

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. The counters are updated at the following occasions:

- **When the process is about to send a message and the trace token is not empty.**

  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 message.
- **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 a message is received and contains a non-empty trace token.**

  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}.
```

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.

close(Continuation) -> ok | {error, Reason}

Types:

Continuation = *continuation()*

Reason = *file:posix()*

Closes a log file properly.

open(Filename) -> *open_ret()*

open(Filename, N) -> *open_ret()*

Types:

```
Filename = string() | atom()
N = integer()
open_ret() =
    {ok, Continuation :: continuation()} |
    {error, Reason :: tuple()}
```

`Filename` specifies the name of the file to be read.

`N` specifies the index of the file to be read. If `N` is omitted, the whole wrap log file is read; if it is specified, only the specified index file is read.

Returns `{ok, Continuation}` if the log/index file is opened successfully. `Continuation` is to be used when chunking or closing the file.

Returns `{error, Reason}` for all errors.

Known Limitations

This version of `wrap_log_reader` does not detect if `disk_log` wraps to a new index file between a call to `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

disk_log(3)

zlib

Erlang module

This module is moved to the *ERTS* application.

app

Name

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**:

```
{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}]}.

             Value                Default
             -----                -------
Application  atom()               -
Description  string()             ""
Id           string()             ""
Vsn          string()             ""
Modules      [Module]             []
MaxP         int()                infinity
MaxT         int()                infinity
Names        [Name]               []
Apps         [App]                []
Env          [{Par,Val}]          []
Start        {Module,StartArgs}   []
Phases       [{Phase,PhaseArgs}]  undefined
RTDeps       [ApplicationVersion] []

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

`Application`
: Application name.

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`

- vsn
- modules
- registered
- applications

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)`*

config

Name

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. A relative path is relative the current working directory of the emulator.

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*