Kernel

Copyright © 1997-8 2011 Ericsson AB. All Rights Reserved.
Kernel 2.14.4
June 8 2011

Copyright © 1997-8 2011 Ericsson AB. All Rights Reserved.
The contents of this file are subject to the Erlang Public License, Version 1.1, (the "License"); you may not use this file except in compliance with the License. You should have received a copy of the Erlang Public License along with this software. If not, it can be retrieved online at http://www.erlang.org/. Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License. Ericsson AB. All Rights Reserved..

June 8 2011

1 Reference Manual

The *Kernel* application has all the code necessary to run the Erlang runtime system itself: file servers and code servers and so on.

kernel

Application

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. The Kernel application contains the following services:

- application controller, see `application(3)`
- `code`
- `disk_log`
- `dist_ac`, distributed application controller
- `erl_boot_server`
- `erl_ddll`
- `error_logger`
- `file`
- `global`
- `global_group`
- `heart`
- `inet`
- `net_kernel`
- `os`
- `pg2`
- `rpc`
- `seq_trace`
- `user`

Error Logger Event Handlers

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

Configuration

The following configuration parameters are defined for the Kernel application. See `app(3)` for more information about configuration parameters.

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

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

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

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

`distributed = [Distrib]`

Specifies which applications are distributed and on which nodes they may 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(3)`, function `load/2`.

`dist_auto_connect = Value`

Specifies when nodes will be automatically connected. If this parameter is not specified, a node is always automatically connected, e.g when a message is to be sent to that node. `Value` is one of:

`never`
Connections are never automatically connected, they must be explicitly connected. See `net_kernel(3)`.

`once`
Connections will be 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(3)`, function `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 the file `FileName`, where `FileName` is a string.

`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.

`global_groups = [GroupTuple]`

Defines global groups, see `global_group(3)`.

- `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 several network interfaces, this parameter specifies which one to listen on. See `inet(3)` for the type definition of `ip_address()`.

`{inet_dist_listen_min, First}`

See below.

`{inet_dist_listen_max, Last}`

Define the `First..Last` port range for the listener socket of a distributed Erlang node.

`inet_parse_error_log = silent`

If this configuration parameter is 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. See ERTS User's Guide, Inet configuration.

`net_setuptime = SetupTime`

`SetupTime` must be a positive integer or floating point number, and will be interpreted as the maximally allowed time for each network operation during connection setup to another Erlang node. The maximum allowed value is 120; if higher values are given, 120 will be used. The default value if the variable is not given, or if the value is incorrect (e.g. not a number), is 7 seconds.

Note 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 given in seconds. Once every `TickTime/4` second, all connected nodes are ticked (if anything else has been written to a node) and if nothing has been received from another node within the last four (4) tick times that node is considered to be down. This ensures that nodes which 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` is by default 60 (seconds). Thus, `45 < T < 75` seconds.

Note: All communicating nodes should have the same `TickTime` value specified.

Note: Normally, a terminating node is detected immediately.

`sync_nodes_mandatory = [NodeName]`

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

`sync_nodes_optional = [NodeName]`

Specifies which other nodes *can* be alive in order 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 amount of time (in milliseconds) this node will wait for the mandatory and optional nodes to start. If this parameter is undefined, no node synchronization is performed. This option also makes sure that `global` is synchronized.

`start_dist_ac = true | false`

Starts the `dist_ac` server if the parameter is `true`. This parameter should be set to `true` for systems that use distributed applications.

The default value is `false`. If this parameter is undefined, the server is started if the 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 should be set to `true` in an embedded system which uses this service.

The default value is `false`.

`boot_server_slaves = [SlaveIP]`

If the `start_boot_server` configuration parameter 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 are:
`'150.236.16.70', "150,236,16,70", {150,236,16,70}`.

The default value is `[]`.

`start_disk_log = true | false`

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

The default value is `false`.

`start_pg2 = true | false`

Starts the `pg2` server (see `pg2(3)`) if the parameter is `true`. This parameter should be set to `true` in an embedded system which uses this service.

The default value is `false`.

`start_timer = true | false`

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

The default value is `false`.

`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.

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)

application

Erlang module

In OTP, *application* denotes a component implementing some specific functionality, that can be started and stopped as a unit, and which can be re-used in other systems as well. This module interfaces the *application controller*, a process started at every Erlang runtime system, and 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* called `Application.app`, where `Application` is the name of the application. Refer to *app(4)* for more information about the application specification.

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 should be located in an *application callback module* exporting a pre-defined set of functions.

Refer to *OTP Design Principles* for more information about applications and behaviours.

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

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

If the specified application is not loaded, or the configuration parameter does not exist, or if the process executing the call does not belong to any application, the function returns `undefined`.

```
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.

If the specified application is not loaded, or the specification key does not exist, or if the process executing the call does not belong to any application, the function returns `undefined`.

```
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()) |
                      {Module :: module(), Version :: term()}]}
                  | {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 will also load the application specifications for any included applications. Note that the function does not load the actual Erlang object code.

The application can be given by its name `Application`. In this case the application controller will search the code path for the application resource file `Application.app` and load the specification it contains.

The application specification can also be given directly as a tuple `AppSpec`. This tuple should have the format and contents as described in `app(4)`.

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

`Nodes` is a list of node names where the application may 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 should preferably be started at `cp1@cave`. If `cp1@cave` is down, the application should be started at either `cp2@cave` or `cp3@cave`.

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

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

Types:

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

Returns a list with information about the applications which have been loaded using `load/1,2`, also included applications. `Application` is the application name. `Description` and `Vsn` are the values of its `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 have been 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` will return `ok` but the application will not be started until the permission is set to `true`.

If the permission of a running application is set to `false`, the application will be stopped. If the permission later is set to `true`, it will be 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` above).

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 may return `ok` even though the application itself has not started. This is true when an application cannot start because it has dependencies to other applications which have not yet been started. When they have been started, `Application` will be started as well.

By default, all applications are loaded with permission `true` on all nodes. The permission is configurable by using the Kernel configuration parameter `permissions`.

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

`set_env(Application, Par, Val, Timeout) -> ok`

Types:

```
Application = Par = atom()
Val = term()
Timeout = timeout()
```

Sets the value of the configuration parameter `Par` for `Application`.

`set_env/3` uses the standard `gen_server` timeout value (5000 ms). A `Timeout` argument can be provided if another timeout value is useful, for example, in situations where the application controller is heavily loaded.

> Warning:
>
> Use this function only if you know what you are doing, that is, on your own applications. It is very application and configuration parameter dependent when and how often the value is read by the application, and careless use of this function may 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 will first load it using `load/1`. It will make sure any included applications are loaded, but will 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 that should be started before this application are running. If not, `{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`.

The `Type` argument 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.

Note that it is always possible to stop an application explicitly by calling `stop/1`. Regardless of the type of the application, no other applications will be affected.

Note also that the transient type is of little practical use, since 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 being started, to determine the start type which is either `StartType` or `local`.

See *`Module:start/2`* for a description of `StartType`.

`local` is returned if only parts of the application is being 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 shutdown (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 itself terminates. Note that all processes with the application master as group leader, i.e. processes spawned from a process belonging to the application, thus are terminated as well.

When stopped, the application is still loaded.

In order to stop a distributed application, `stop/1` has to 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 will stop its execution. The application will not be moved between nodes due to `stop/1` being 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()

Performs a takeover of 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, i.e. when `Module:start/2` and any calls to `Module:start_phase/3` have returned.

Thus two instances of the application will run simultaneously during the takeover, which makes it possible to transfer data from the old to the new instance. If this is not acceptable behavior, parts of the old instance may be shut down when the new instance is started. Note that the application may not be stopped entirely however, at least the top supervisor must remain alive.

See `start/1,2` for a description of `Type`.

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

Types:

Application = atom()

Reason = term()

Unloads the application specification for `Application` from the application controller. It will also unload the application specifications for any included applications. Note that the function does not purge the actual Erlang object code.

unset_env(Application, Par) -> ok

unset_env(Application, Par, Timeout) -> ok

Types:

```
Application = Par = atom()
Timeout = timeout()
```

Removes the configuration parameter `Par` and its value for `Application`.

`unset_env/2` uses the standard `gen_server` timeout value (5000 ms). A `Timeout` argument can be provided if another timeout value is useful, for example, in situations where the application controller is heavily loaded.

> ### Warning:
>
> Use this function only if you know what you are doing, that is, on your own applications. It is very application and configuration parameter dependent when and how often the value is read by the application, and careless use of this function may 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 which are currently running. `Application` is the application name. `Description` and `Vsn` are the values of its `description` and `vsn` application specification keys, respectively.

`which_applications/0` uses the standard `gen_server` timeout value (5000 ms). A `Timeout` argument can be provided if another timeout value is useful, for example, in situations where the application controller is heavily loaded.

CALLBACK MODULE

The following functions should be exported from an `application` callback module.

Exports

```
Module:start(StartType, StartArgs) -> {ok, Pid} | {ok, Pid, State} | {error,
Reason}
```

Types:

StartType = normal | {takeover,Node} | {failover,Node}

Node = node()

StartArgs = term()

Pid = pid()

State = term()

This function is called whenever an application is started using `application:start/1,2`, and should 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's a normal startup.
- `normal` also if the application is distributed and started at the current node due to 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 due to a takeover from `Node`, either because `application: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 due to 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 should 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 later the application is stopped, `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()**

This function is used to start an application with included applications, when there is a need for synchronization between processes in the different applications during startup.

The start phases is 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.

See `Module:start/2` for a description of `StartType`.

`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 will be passed to `Module:stop/1`.

The function is optional. If it is not defined, the processes will be 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 should 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 there are any changes to the configuration parameters.

`Changed` is a list of parameter-value tuples with all configuration parameters with changed values, `New` is a list of parameter-value tuples with all configuration parameters that have been added, and `Removed` is a list of all parameters that have been removed.

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

`is_auth(Node) -> yes | no`

Types:

`Node = node()`

Returns `yes` if communication with `Node` is authorized. Note that a connection to `Node` will be 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.

`cookie() -> Cookie`

Types:

`Cookie = cookie()`

Use *erlang:get_cookie()* instead.

`cookie(TheCookie) -> true`

Types:

`TheCookie = Cookie | [Cookie]`

The cookie may also be given as a list with a single atom element.

`Cookie = cookie()`

Use *erlang:set_cookie(node(), Cookie)* 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 either *embedded* or *interactive* mode. Which one is decided by the command line flag `-mode`.

```
% erl -mode interactive
```

Default mode is `interactive`.

- In embedded mode, all code is loaded during system start-up according to the boot script. (Code can also be loaded later by explicitly ordering the code server to do so).
- In interactive mode, only some code is loaded during system startup-up, basically the modules needed by the runtime system itself. 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 modules affecting the Erlang runtime system itself, the `kernel`, `stdlib` and `compiler` directories 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 the 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 the 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 which have the same `Name`. The `-Vsn` suffix is optional. If an `ebin` directory exists under `Name[-Vsn]`, it is this directory which is added to the code path.

The environment variable `ERL_LIBS` (defined in the operating system) can be used to define additional library directories that will be handled in the same way as the standard OTP library directory described above, except that directories that do not have an `ebin` directory will be ignored.

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

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

Example: On an Unix-like system, `ERL_LIBS` could be set to `/usr/local/jungerl:/home/some_user/my_erlang_lib`. (On Windows, use semi-colon as separator.)

Code Path Cache

The code server incorporates a code path cache. The cache functionality is disabled by default. To activate it, start the emulator with the command line flag `-code_path_cache` or call `code:rehash()`. When the cache is created

(or updated), the code server searches for modules in the code path directories. This may take some time if the the code path is long. After the cache creation, the time for loading modules in a large system (one with a large directory structure) is significantly reduced compared to having the cache disabled. The code server is able to look up the location of a module from the cache in constant time instead of having to search through the code path directories.

Application resource files (`.app` files) are also stored in the code path cache. This feature is used by the application controller (see *application(3)*) to load applications efficiently in large systems.

Note that when the code path cache is created (or updated), any relative directory names in the code path are converted to absolute.

Loading of Code From Archive Files

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

In the current implementation, Erlang archives are `ZIP` files with `.ez` extension. Erlang archives may also be enclosed in `escript` files whose file extension is arbitrary.

Erlang archive files may 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 would 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 with the name `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 may 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 may be compressed, but in order to speed up the access of frequently read files, it may be a good idea to store `beam` and `app` files uncompressed in the archive.

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

The code server uses the module `erl_prim_loader` (possibly via the `erl_boot_server`) to read code files from archives. But the functions in `erl_prim_loader` may 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 may coexist. This may be useful when there is a need of having parts of the application as regular files. A typical case is the `priv` directory which must reside as a regular directory in order to be able to dynamically link in drivers and start port programs. For other applications that do not

have this need, the `priv` directory may reside in the archive and the files under the `priv` directory may be read via the `erl_prim_loader`.

At the time point when a directory is added to the code path as well as when the entire code path is (re)set, the code server will decide which subdirectories in an application that shall be read from the archive and which that shall be read as regular files. If directories are added or removed afterwards, the file access may fail if the code path is not updated (possibly to the same path as before in order to trigger the directory resolution update). For each directory on the second level (ebin, priv, src etc.) in the application archive, the code server will firstly choose the regular directory if it exists and secondly from the archive. The function `code:lib_dir/2` returns the path to the subdirectory. For example `code:lib_dir(megaco,ebin)` may return `/otp/root/lib/megaco-3.9.1.1.ez/megaco-3.9.1.1/ebin` while `code:lib_dir(megaco,priv)` may return `/otp/root/lib/megaco-3.9.1.1/priv`.

When an `escript` file contains an archive, there are neither restrictions on the name of the `escript` nor on how many applications that may be stored in the embedded archive. Single `beam` files may also reside on the top level in the archive. At startup, both the top directory in the embedded archive as well as all (second level) `ebin` directories in the embedded archive are added to the code path. See *escript(1)*

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

This behavior can be controlled via the command line flag `-code_path_choice Choice`. If the flag is set to `relaxed`, the code server will instead choose a suitable directory depending on the actual file structure. If there exists a regular application ebin directory,situation it will be choosen. But if it does not exist, the ebin directory in the archive is choosen if it exists. If neither of them exists the original directory will be choosen.

The command line flag `-code_path_choice Choice` does also affect how `init` interprets the `boot script`. The interpretation of the explicit code paths in the `boot script` may be `strict` or `relaxed`. It is particular useful to set the flag to `relaxed` when you want to elaborate with code loading from archives without editing the `boot script`. The default is `relaxed`. See *init(3)*

Current and Old Code

The code of a module can exists in two variants in a system: *current code* and *old code*. When a module is loaded into the system for the first time, the code of the module 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 (perhaps because of the correction of an error), then 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 if it was loaded for the first time, as described above, and becomes 'current'.

Both old and current code for a module are valid, and may even be evaluated concurrently. The difference is that exported functions in old code are unavailable. Hence there is no way to make a global call to an exported function in old code, but old code may still be evaluated because of processes lingering in it.

If a third instance of the module is loaded, the code server will remove (purge) the old code and any processes lingering in it will be 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, refer to *Erlang Reference Manual*.

Argument Types and Invalid Arguments

Generally, 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.

From the R12B release, functions in this module will generally fail with an exception if they are passed an incorrect type (for instance, an integer or a tuple where an atom was expected). An error tuple will be returned if type of argument was correct, but there was some other error (for instance, a non-existing directory given to `set_path/1`.

Data Types

```
load_error_rsn() = badfile
                 | native_code
                 | nofile
                 | not_purged
                 | sticky_directory
```

Exports

`set_path(Path) -> true | {error, What}`

Types:

Path = **[Dir]**

Dir = **string()**

What = **bad_directory** | **bad_path**

Sets the code path to the list of directories `Path`.

Returns `true` if successful, or `{error, bad_directory}` if any `Dir` is not the name of a directory, or `{error, bad_path}` if the argument is invalid.

`get_path() -> Path`

Types:

Path = **[Dir]**

Dir = **string()**

Returns the code path

`add_path(Dir) -> true | {error, What}`

`add_pathz(Dir) -> true | {error, What}`

Types:

Dir = **string()**

What = **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) -> true | {error, What}`

Types:

Dir = **string()**

What = **bad_directory**

Adds `Dir` to the beginning of the code path. If `Dir` already 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]

Dir = string()

Adds the directories in `Dirs` to the end of the code path. If a `Dir` already exists, it is not added. This function always returns `ok`, regardless of the validity of each individual `Dir`.

add_pathsa(Dirs) -> ok

Types:

Dirs = [Dir]

Dir = string()

Adds the directories in `Dirs` to the beginning of the code path. If a `Dir` already exists, it is removed from the old position in the code path. This function always returns `ok`, regardless of the validity of each individual `Dir`.

del_path(Name | Dir) -> true | false | {error, What}

Types:

Name = atom()

Dir = string()

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. It is also possible to give the complete directory name `Dir` as argument.

Returns `true` if successful, or `false` if the directory is not found, or `{error, bad_name}` if the argument is invalid.

replace_path(Name, Dir) -> true | {error, What}

Types:

Name = atom()

Dir = string()

What = bad_name | bad_directory | {badarg, term()}

This function 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 should be used if a new version of the directory (library) is added to a running system.

Returns `true` if successful, or `{error, bad_name}` if `Name` is not found, or `{error, bad_directory}` if `Dir` does not exist, or `{error, {badarg, [Name, Dir]}}` if `Name` or `Dir` is invalid.

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

Types:

Module = atom()

What = nofile | sticky_directory | badarg | term()

Tries to load the Erlang module `Module`, using the code path. It looks for the object code file with an extension that corresponds 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, nofile}` if no object code is found, or `{error, sticky_directory}` if the object code resides in a sticky directory, or `{error, badarg}` if the argument is invalid. Also if the loading fails, an error tuple is returned. See *erlang:load_module/2* for possible values of `What`.

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

Types:

Filename = string()

Module = atom()

What = nofile | sticky_directory | badarg | term()

Does the same as `load_file(Module)`, but `Filename` is either an absolute file name, or a relative file name. The code path is not searched. It returns a value in the same way as *load_file/1*. Note that `Filename` should not contain the extension (for example `".beam"`); `load_abs/1` adds the correct extension itself.

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

Types:

Module = atom()

What = nofile | sticky_directory | embedded | badarg | term()

Tries to to load a module in the same way as *load_file/1*, unless the module is already loaded. In embedded mode, however, it does not load a module which is not already loaded, but returns `{error, embedded}` instead.

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

Types:

Module = atom()

Filename = string()

What = sticky_directory | badarg | term()

This function can be used to load object code on remote Erlang nodes. The 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. Accordingly, `Filename` is not opened and read by the code server.

Returns `{module, Module}` if successful, or `{error, sticky_directory}` if the object code resides in a sticky directory, or `{error, badarg}` if any argument is invalid. Also if the loading fails, an error tuple is returned. See *erlang:load_module/2* for possible values of `What`.

`delete(Module) -> true | false`

Types:

Module = atom()

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 that no external function calls can be made to it.

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

purge(Module) -> true | false

Types:

Module = atom()

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.

Returns `true` if successful and any process needed to be killed, otherwise `false`.

soft_purge(Module) -> true | false

Types:

Module = atom()

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

Returns `false` if the module could not be purged due to processes lingering in old code, otherwise `true`.

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

Types:

Module = atom()

Loaded = Absname | preloaded | cover_compiled

Absname = string()

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

Normally, `Loaded` is the absolute file name `Absname` from which the code was 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 = atom()

Loaded = Absname | preloaded | cover_compiled

Absname = string()

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

which(Module) -> Which

Types:

Module = atom()

Which = Filename | non_existing | preloaded | cover_compiled

Filename = string()

If the module is not loaded, this function searches the code path for the first file which contains object code for `Module` and returns the absolute file name. If the module is loaded, it returns the name of the file which contained the loaded object code. If the module is pre-loaded, `preloaded` is returned. If the module is Cover compiled, `cover_compiled` is returned. `non_existing` is returned if the module cannot be found.

get_object_code(Module) -> {Module, Binary, Filename} | error

Types:

Module = atom()

Binary = binary()

Filename = string()

Searches the code path for the object code of the module `Module`. It returns `{Module, Binary, Filename}` if successful, and `error` if not. `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() -> string()`

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

```
> code:root_dir().
"/usr/local/otp"
```

`lib_dir() -> string()`

Returns the library directory, `$OTPROOT/lib`, where `$OTPROOT` is the root directory of Erlang/OTP.

```
> code:lib_dir().
"/usr/local/otp/lib"
```

`lib_dir(Name) -> string() | {error, bad_name}`

Types:

Name = atom()

This function is mainly intended for finding out the path for the "library directory", the top directory, for an application `Name` located under `$OTPROOT/lib` or on a directory referred to via the `ERL_LIBS` environment variable.

If there is a regular directory called `Name` or `Name-Vsn` 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 the 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` will be returned. This means that the library directory for an application is the same, regardless of whether the application resides in an archive or not.

```
> 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 via the `ERL_LIBS` environment variable. 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) -> string() | {error, bad_name}`

Types:

Name = atom()

SubDir = atom()

Returns the path to a subdirectory directly under the top directory of an application. Normally the subdirectories resides 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 may reside as regular directories while other resides in an archive file. It is not checked if this directory really exists.

```
> 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() -> string()`

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

`priv_dir(Name) -> string() | {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() -> ".beam"`

Returns the object code file extension that corresponds to the Erlang machine used, namely `".beam"`.

`stick_dir(Dir) -> ok | error`

Types:

Dir = string()

What = term()

This function marks `Dir` as sticky.

Returns `ok` if successful or `error` if not.

`unstick_dir(Dir) -> ok | error`

Types:

Dir = string()

What = term()

This function unsticks a directory which has been marked as sticky.

Returns `ok` if successful or `error` if not.

`is_sticky(Module) -> true | false`

Types:

Module = atom()

This function returns `true` if `Module` is the name of a module that has been loaded from a sticky directory (or in other words: an attempt to reload the module will fail), or `false` if `Module` is not a loaded module or is not sticky.

`rehash() -> ok`

This function creates or rehashes the code path cache.

`where_is_file(Filename) -> Absname | non_existing`

Types:

Filename = Absname = string()

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. If the code path cache is used, the code server will efficiently read the full name from the cache, provided that `Filename` is an object code file or an `.app` file.

`clash() -> ok`

Searches the entire code space for module names with identical names and writes a report to `stdout`.

`is_module_native(Module) -> true | false | undefined`

Types:

Module = atom()

This function returns `true` if `Module` is name of a loaded module that has native code loaded, and `false` if `Module` is loaded but does not have native. If `Module` is not loaded, this function returns `undefined`.

disk_log

Erlang module

`disk_log` is a disk based term logger which makes it possible to efficiently log items on files. Two types of logs are supported, *halt logs* and *wrap logs*. A halt log appends items to a single file, the size of which may or may not be limited by the disk log module, whereas a wrap log utilizes a sequence of wrap log files of limited size. As a wrap log file has been filled up, further items are logged onto to the next file in the sequence, starting all over with the first file when the last file has been filled up. For the sake of efficiency, items are always written to files as binaries.

Two formats of the log files are supported, the *internal format* and the *external format*. The internal format supports automatic repair of log files that have not been properly closed, and makes it possible to efficiently read logged items in *chunks* using a set of functions defined in this module. In fact, this is the only way to read internally formatted logs. The external format leaves it up to the user to read the logged deep byte lists. The disk log module cannot repair externally 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).

For each open disk log there is one process that handles requests made to the disk log; the disk log 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 either be an *owner* or an anonymous *user* of the disk log. Each owner is linked to the disk log process, and the disk log is closed by the owner should the owner terminate. Owners can subscribe to *notifications*, messages of the form `{disk_log, Node, Log, Info}` that are sent from the disk log process when certain events occur, see the commands below and in particular the `open/1` option *notify*. There can be several owners of a log, but a process cannot own a log more than once. One and the same process may, however, open the log as a user more than once. For a disk log process to properly close its file 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 the 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 have been 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 actually write the items to the file, but return the control to the caller more or less immediately.

When using the internal format for logs, the functions `log/2`, `log_terms/2`, `alog/2`, and `alog_terms/2` should be used. 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 deep lists of bytes or, alternatively, binaries of deep lists of bytes. For example, to log the string `"hello"` in ASCII format, we 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 be used for internally formatted logs as well, 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 will fail. The corresponding terms (not the binaries) will be 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 *a distributed disk log* if requests made to any one of the logs are automatically made to the other logs as well. The members of such a collection will be 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 instance, logged terms are not necessarily written onto the node where the request was made before written onto the other nodes. One could note here that there are a few functions that do not make requests to all members of distributed disk logs, namely `info`, `chunk`, `bchunk`, `chunk_step` and `lclose`. An open disk log that is not a distributed disk log is said to be a *local disk log*. A local disk log is accessible only from the node where the disk log process runs, whereas a distributed disk log is accessible from all nodes in the Erlang system, with exception 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; there is no attempt 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 will be 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 may not be acceptable that replies from individual logs are ignored. An alternative in such situations is to use several 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 the disk log module replies with an error message, but when called asynchronously, the disk log module does not know where to send the error message. Instead owners subscribing to notifications will receive an `error_status` message.

The disk log module itself does not report errors to the `error_logger` module; it is up to the caller to decide whether the error logger should be employed or not. The function `format_error/1` can be used to produce readable messages from error replies. Information events are however sent to the error logger in two situations, namely when a log is repaired, or when a file is missing while reading chunks.

The error message `no_such_log` means that the given disk log is not currently 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` and `truncate`) the disk log process immediately terminates. Before the process terminates links to to owners and blocking processes (see `block`) are removed. The effect is that the links work in one direction only; any process using a disk log has to check for the error message `no_such_log` if some other process might truncate or reopen 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() | binary() | [*dlog_byte()*]

dlog_byte() = [*dlog_byte()*] | byte()

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`.

bytes() = binary() | [byte()]

invalid_header() = term()

file_error() = term()

Exports

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

Types:

LocalLog = DistributedLog = *log()*

The `accessible_logs/0` function 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 = *bytes()*
notify_ret() = ok | {error, no_such_log}

The `alog/2` and `balog/2` functions asynchronously append an item to a disk log. The function `alog/2` is used for internally formatted logs, and the function `balog/2` for externally formatted logs. `balog/2` can be used for internally formatted logs as well provided the binary was constructed with a call to `term_to_binary/1`.

The owners that subscribe to notifications will receive the message `read_only`, `blocked_log` or `format_external` in case the item cannot be written on the log, and possibly one of the messages `wrap`, `full` and `error_status` if an item was written on the log. The message `error_status` is sent if there is something wrong with the header function or a file error occurred.

alog_terms(Log, TermList) -> *notify_ret()*

balog_terms(Log, ByteList) -> *notify_ret()*

Types:

Log = *log()*
TermList = [term()]
ByteList = [*bytes()*]
notify_ret() = ok | {error, no_such_log}

The `alog_terms/2` and `balog_terms/2` functions asynchronously append a list of items to a disk log. The function `alog_terms/2` is used for internally formatted logs, and the function `balog_terms/2` for externally formatted logs. `balog_terms/2` can be used for internally formatted logs as well provided the binaries were constructed with calls to `term_to_binary/1`.

The owners that subscribe to notifications will receive the message `read_only`, `blocked_log` or `format_external` in case the items cannot be written on the log, and possibly one or more of the messages `wrap`, `full` and `error_status` if items were written on the log. The message `error_status` is sent if there is something wrong with the header function or a file error occurred.

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 is used to ensure that the disk log is unblocked should the blocking process terminate 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 the functions `chunk/2,3`, `bchunk/2,3`, `chunk_step/3`, and `unblock/1` without being affected by the block. Any other attempt than those hitherto mentioned to update or read a blocked log suspends the calling process until the log is unblocked or returns an error message `{blocked_log, Log}`, depending on whether the value of `QueueLogRecords` is `true` or `false`. The default value of `QueueLogRecords` is `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()}
Reason = no_such_log
       | nonode
       | {read_only_mode, Log}
       | {blocked_log, Log}
       | {badarg, head}
```

The `change_header/2` function changes the value of the `head` or `head_func` option 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}
```

The `change_notify/3` function changes the value of the `notify` option 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, CurrentSize :: integer() >= 1}
       | {badarg, size}
       | {file_error, file:filename(), file_error()}
```

The `change_size/2` function changes the size of an open log. For a halt log it is always possible to increase the size, but it is not possible to decrease the size to something less than the current size of the file.

For a wrap log it is always possible to increase both the size and number of files, as long as the number of files does not exceed 65000. If the maximum number of files is decreased, the change will not be valid until the current file is full and the log wraps to the next file. The redundant files will be removed next time the log wraps around, i.e. 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, the files 7 to 10 will be removed when file number 6 is full and the log starts to write to file number 1 again. Otherwise the files greater than the current file will be removed when the current file is full (e.g. if the current file is 8, the files 9 and 10); the files between new maximum number of files and the current file (i.e. files 7 and 8) will be removed next time file number 6 is full.

If the size of the files is decreased the change will immediately affect the current log. It will not of course change the size of log files already full until next time they are used.

If the log size is decreased for instance to save space, the 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()}
```

The `chunk/2,3` and `bchunk/2,3` functions make it possible to efficiently read the terms which have been appended to an internally formatted log. It minimizes disk I/O by reading 64 kilobyte chunks from the file. The `bchunk/2,3` functions return the binaries read from the file; they do not call `binary_to_term`. Otherwise the work just like `chunk/2,3`.

The first time `chunk` (or `bchunk`) is called, an initial continuation, the atom `start`, must be provided. If there is a disk log process 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. Default is `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 has been reached.

The `chunk` function 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` it is possible to extract all terms from a log.

The `chunk` function 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 which were found not to be Erlang

terms in the chunk. Note also that the log is not repaired. When trying to read chunks from a log opened in read-write mode, the 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. Should a wrap log file be missing, a message is output on the error log.

When `chunk/2,3` is used with wrap logs, the returned continuation may or may not be valid in the next call to `chunk`. This is because the log may wrap and delete the file into which the continuation points. To make sure this does not happen, 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}
```

The `chunk_info/1` function returns the following pair describing the chunk continuation returned by `chunk/2,3`, `bchunk/2,3`, or `chunk_step/3`:

- `{node, Node}`. 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()}
```

The function `chunk_step` can be used in conjunction 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 the atom `start` is given 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 have not been used yet, `{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()}
```

The function `close/1` 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 will be activated next time the log is opened.

The disk log process in not terminated as long as there are owners or users of the log. It should be stressed that each and every owner must close the log, possibly by terminating, and that 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 simply 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, the function `format_error` returns a descriptive string of the error in English. For file errors, the function `format_error/1` in the `file` module 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()
```

The `inc_wrap_file/1` function forces the internally formatted disk log to start logging to the next log file. It can be used, for instance, in conjunction with `change_size/2` to reduce the amount of disk space allocated by the disk log.

The owners that subscribe to notifications will normally receive a `wrap` message, but in case of an error with a reason tag of `invalid_header` or `file_error` an `error_status` message will be 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()}
| {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}}
```

The `info/1` function returns a list of `{Tag, Value}` pairs describing the log. If there is a disk log process 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}`, where `Log` is the name of the log as given 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}`, where `Type` is the type of the log as given by the `open/1` option `type`.
- `{format, Format}`, where `Format` is the format of the log as given by the `open/1` option `format`.
- `{size, Size}`, where `Size` is the size of the log as given 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}`, where `Mode` is the mode of the log as given by the `open/1` option `mode`.
- `{owners, [{pid(), Notify}]}` where `Notify` is the value set by the `open/1` option `notify` or the function `change_notify/3` for the owners of the log.
- `{users, Users}` where `Users` is the number of anonymous users of the log, see the `open/1` option *linkto*.
- `{status, Status}`, where `Status` is `ok` or `{blocked, QueueLogRecords}` as set by the functions `block/1,2` and `unblock/1`.
- `{node, Node}`. The information returned by the current invocation of the `info/1` function has been gathered from the disk log process running on `Node`.
- `{distributed, Dist}`. If the log is local on the current node, then `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 of the value of the `open/1` options `head` and `head_func` or set by the 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}`, where `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}`, where `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}` is the number of bytes written to the current wrap log file.
- `{no_current_items, integer() >= 0}` is the number of items written to the current wrap log file, header inclusive.
- `{no_items, integer() >= 0}` is the total number of items in all wrap log files.
- `{current_file, integer()}` is the ordinal for the current wrap log file in the range `1..MaxNoFiles`, where `MaxNoFiles` is given by the `open/1` option `size` or set by `change_size/2`.
- `{no_overflows, {SinceLogWasOpened, SinceLastInfo}}`, where `SinceLogWasOpened` (`SinceLastInfo`) is the number of times a wrap log file has been filled up and a new one 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.

Note that the `chunk/2,3`, `bchunk/2,3`, and `chunk_step/3` functions 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()}
```

The function `lclose/1` closes a local log or an individual distributed log on the current node. The function `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 there is no log with the given name 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 = bytes()
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()}
```

The `log/2` and `blog/2` functions synchronously append a term to a disk log. They return `ok` or `{error, Reason}` when the term has been written to disk. If the log is distributed, `ok` is always returned, unless all nodes are down. Terms are written by means of the ordinary `write()` function of the operating system. Hence, there is no

guarantee that the term has actually been written to the disk, it might linger in the operating system kernel for a while. To make sure the item is actually written to disk, the `sync/1` function must be called.

The `log/2` function is used for internally formatted logs, and `blog/2` for externally formatted logs. `blog/2` can be used for internally formatted logs as well provided the binary was constructed with a call to `term_to_binary/1`.

The owners that subscribe to notifications will be 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 = [bytes()]
   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()}
```

The `log_terms/2` and `blog_terms/2` functions synchronously append a list of items to the log. The benefit of using these functions rather than the `log/2` and `blog/2` functions is that of efficiency: the given 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.

The `log_terms/2` function is used for internally formatted logs, and `blog_terms/2` for externally formatted logs. `blog_terms/2` can be used for internally formatted logs as well provided the binaries were constructed with calls to `term_to_binary/1`.

The owners that subscribe to notifications will be 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()}
              | {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}
```

The `ArgL` parameter is a list of options which have the following meanings:

- `{name, Log}` specifies the name of the log. This is the name which 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 which will be used for logged terms. If this value is omitted and the name of the log is either an atom or a string, the file name will default to `lists:concat([Log, ".LOG"])` for halt logs. For wrap logs, this will be the base name of the files. Each file in a wrap log will be called `<base_name>.N`, where `N` is an integer. Each wrap log will also have two files called `<base_name>.idx` and `<base_name>.siz`.
- `{linkto, LinkTo}`. If `LinkTo` is a pid, that pid 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 which calls `open/1` owns the log.
- `{repair, Repair}`. If `Repair` is `true`, the current log file will be repaired, if needed. As the restoration is initiated, a message is output on the error log. If `false` is given, no automatic repair will be attempted. Instead, the tuple `{error, {need_repair, Log}}` is returned if an attempt is made to open a corrupt log file. If `truncate` is given, the log file will be truncated, creating an empty log. Default is `true`, which has no effect on logs opened in read-only mode.
- `{type, Type}` is the type of the log. Default is `halt`.
- `{format, Format}` specifies the format of the disk log. Default is `internal`.
- `{size, Size}` specifies the size of the log. When a halt log has reached its maximum size, all attempts to log more items are rejected. The default size is `infinity`, which for halt implies that there is no maximum size. For wrap logs, the `Size` parameter may be either a pair `{MaxNoBytes, MaxNoFiles}` or `infinity`. In the latter case, if the files of an already 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 is 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 the option `Size`, but any supplied value must equal the current size of the log, 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. The default value is `[]`, which means that the log is local on the current node.
- `{notify, bool()}`. If `true`, the owners of the log are notified when certain events occur in the log. Default is `false`. The owners are sent one of the following messages when an event occurs:
  - `{disk_log, Node, Log, {wrap, NoLostItems}}` is 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 have been lost when truncating existing files.
  - `{disk_log, Node, Log, {truncated, NoLostItems}}` is sent when a log has been 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}}` is 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}}` is 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}}` is sent when `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}` is sent when an attempt to log items to a wrap log would write more bytes than the limit set by the `size` option.
  - `{disk_log, Node, Log, {error_status, Status}}` is sent when the error status changes. The error status is defined by the outcome of the last attempt to log items to a the log or to truncate the log or the last use of `sync/1`, `inc_wrap_file/1` or `change_size/2`. `Status` is one of `ok` and `{error, Error}`, the former being 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` should be a term if the format is `internal`, and a deep list of bytes (or a binary) otherwise. Default is `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` should be a term if the format is `internal`, and a deep list of bytes (or a binary) otherwise.
- `{mode, Mode}` specifies if the log is to be opened in read-only or read-write mode. It defaults to `read_write`.

The `open/1` function returns `{ok, Log}` if the log file was successfully opened. If the file was 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 which were non-Erlang terms. If the `distributed` parameter was given, `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 will commence 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.

The `open/1` function cannot be used for changing the values of options of an already open log; when there are prior owners or users of a log, all option values except `name`, `linkto` and `notify` are just checked against the values that have been supplied before as option values to `open/1`, `change_header/2`, `change_notify/3` or `change_size/2`. As a consequence, none of the options except `name` is mandatory. If some given value differs from the current value, a tuple `{error, {arg_mismatch, OptionName, CurrentValue, Value}}` is returned. Caution: an owner's attempt to open a log as owner once again is acknowledged with the return value `{ok, Log}`, but the state of the disk log is not affected in any way.

If a log with a given name is local on some node, and one tries to open the log distributed on the same node, then 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 option values supplied are used on all nodes mentioned by the `distributed` option. 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 several calls to `open/1`.

It is possible to open a log file more than once by giving different values to the option `name` or by using the same file when distributing a log on different nodes. It is up to the user of the `disk_log` module to ensure that no more than one disk log process has write access to any file, or the the file may 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()`*

The `pid2name/1` function returns the name of the log given the pid of a disk log process on the current node, or `undefined` if the given 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 = bytes()
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()}
```

The `reopen` functions first rename the log file to `File` and then re-create a new log file. In case of a wrap log, `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 the `Head` or the `BHead` argument is given, this item is used instead. The header argument is used once only; next time a wrap log file is opened, the header given to `open/1` is used.

The `reopen/2,3` functions are used for internally formatted logs, and `breopen/3` for externally formatted logs.

The owners that subscribe to notifications will 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}]}`, and other processes that have 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()}
```

The `sync/1` function ensures that the contents of the log are actually 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 = bytes()
trunc_error_rsn() = no_such_log
                  | nonode
                  | {read_only_mode, log()}
                  | {blocked_log, log()}
                  | {invalid_header, invalid_header()}
                  | {file_error, file:filename(), file_error()}
```

The `truncate` functions remove all items from a disk log. If the `Head` or the `BHead` argument is given, this item is written first in the newly truncated log, otherwise the header given to `open/1` is used. The header argument is only used once; next time a wrap log file is opened, the header given to `open/1` is used.

The `truncate/1,2` functions are used for internally formatted logs, and `btruncate/2` for externally formatted logs.

The owners that subscribe to notifications will 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}]}`, and other processes that have 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()}
```

The `unblock/1` function 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 which 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 the `-loader inet` command line flag. All hosts specified with the `-hosts Host` command line flag 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 both read regular files as well as files in archives. See *code(3)* and *erl_prim_loader(3)*.

> ### Warning:
>
> The support for loading of code from archive files is experimental. The sole purpose of releasing it before it is ready is to obtain early feedback. The file format, semantics, interfaces etc. may be changed in a future release.

Exports

`start(Slaves) -> {ok, Pid} | {error, What}`

Types:

```
Slaves = [Host]
Host = atom()
Pid = pid()
What = any()
```

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, What}`

Types:

```
Slaves = [Host]
Host = atom()
Pid = pid()
What = any()
```

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.

`add_slave(Slave) -> ok | {error, What}`

Types:

```
Slave = Host
Host = atom()
What = any()
```

Adds a `Slave` node to the list of allowed slave hosts.

`delete_slave(Slave) -> ok | {error, What}`

Types:

```
Slave = Host
Host = atom()
What = any()
```

Deletes a `Slave` node from the list of allowed slave hosts.

`which_slaves() -> Slaves`

Types:

```
Slaves = [Host]
Host = atom()
```

Returns the current list of allowed slave hosts.

SEE ALSO

init(3), erl_prim_loader(3)

erl_ddll

Erlang module

The `erl_ddll` 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 the module, as well as for most real world applications, the descriptions of the functions *load/2* and *unload/1* are enough to get going.

The driver should be provided as a dynamically linked library in a object code format specific for the platform in use, i. e. `.so` files on most Unix systems and `.ddl` files on windows. An erlang linked in driver has to provide specific interfaces to the emulator, so this module is not designed for loading arbitrary dynamic libraries. For further information about erlang drivers, refer to the ERTS reference manual section *erl_driver*.

When describing a set of functions, (i.e. 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*. There can be several users in one process (different modules needing the same driver) and several processes running the same code, making up several *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 as well as the number of loads by each process, so that the driver will only be unloaded when no one wants it (it has no user). The driver also keeps track of ports that are opened towards it, so that one can delay unloading until all ports are closed or kill all ports using 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:

Load and unload on a "when needed basis"

This (most common) scenario simply supports that each *user* of the driver loads it when it is needed and unloads it when the *user* no longer have any use for it. 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 really concerned with if the driver is already loaded from the filesystem or if the object code has to be loaded from filesystem.

Two pairs of functions support this scenario:

load/2 and unload/1

When using the `load/unload` interfaces, the driver will not *actually* get unloaded until the *last port* using the driver is closed. The function `unload/1` can return immediately, as the *users* are not really concerned with when the actual unloading occurs. The driver will actually get unloaded when no one needs it any longer.

If a process having the driver loaded dies, it will have the same effect as if unloading was done.

When loading, the function `load/2` returns `ok` as soon as there is any instance of the driver present, so that if a driver is waiting to get unloaded (due to open ports), it will simply change state to no longer need unloading.

load_driver/2 and unload_driver/1

These interfaces is intended to be used when it is considered an error that ports are open towards a driver that no *user* has loaded. The ports still open when the last *user* calls `unload_driver/1` or when the last process having the driver loaded dies, will get 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 occurs when the driver code might need replacement during operation of the Erlang emulator. Implementing driver code replacement is somewhat 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 actual unloading/loading is done as one atomic operation, blocking all processes in the system from using the driver concerned 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 start, 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 than one *user* has the driver loaded.

The two functions for reloading drivers should 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 should be done after the last open port towards the driver is closed.

As `reload/2` actually waits for the reloading to occur, a misbehaving process keeping open ports towards the driver (or keeping the driver loaded) might cause infinite waiting for reload. Timeouts has to be provided outside of the process demanding the reload or by using the low-level interface *try_load/3* in combination with driver monitors (see below).

load_driver/2 and reload_driver/2

This pair of functions are used when open ports towards the driver should be killed with reason `driver_unloaded` to allow for new driver code to get loaded.

If, however, another process has the driver loaded, calling `reload_driver` returns the error code `pending_process`. As stated earlier, the recommended design is to not allow other *users* than the "driver reloader" to actually demand loading of the concerned driver.

Data Types

`driver() = string() | atom()`

`path() = string() | atom()`

Exports

`demonitor(MonitorRef) -> ok`

Types:

MonitorRef = reference()

Removes a driver monitor in much the same way as *erlang:demonitor/1* does with process monitors. See *monitor/2*, *try_load/3* and *try_unload/2* for details about how to create driver monitors.

The function throws a `badarg` exception if the parameter is not a reference().

`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 being a tuple list containing all information available about a driver.

The different tags that will appear in the list are:

- processes
- driver_options
- port_count
- linked_in_driver
- permanent
- awaiting_load
- awaiting_unload

For a detailed description of each value, please read the description of *info/2* below.

The function throws a `badarg` exception if the driver is not present in the system.

`info(Name, Tag) -> Value`

Types:

Name = string() | atom()

Tag = processes | driver_options | port_count | linked_in_driver | permanent | awaiting_load | awaiting_unload

Value = term()

This function returns specific information about one aspect of a driver. The `Tag` parameter specifies which aspect to get information about. The `Value` return differs between different tags:

processes

Return all processes containing *users* of the specific drivers as a list of tuples `{pid(),int()}`, where the `int()` denotes the number of users in the process `pid()`.

driver_options

Return a list of the driver options provided when loading, as well as any options set by the driver itself during initialization. The currently only valid option being `kill_ports`.

port_count

Return the number of ports (an `int()`) using the driver.

linked_in_driver

Return a `bool()`, being `true` if the driver is a statically linked in one and `false` otherwise.

permanent

Return a `bool()`, being `true` if the driver has made itself permanent (and is *not* a statically linked in driver). `false` otherwise.

awaiting_load

Return a list of all processes having monitors for `loading` active, each process returned as `{pid(),int()}`, where the `int()` is the number of monitors held by the process `pid()`.

awaiting_unload

Return a list of all processes having monitors for `unloading` active, each process returned as `{pid(),int()}`, where the `int()` is the number of monitors held by the process `pid()`.

If the options `linked_in_driver` or `permanent` return true, all other options will return the value `linked_in_driver` or `permanent` respectively.

The function throws a `badarg` exception if the driver is not present in the system or 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. The `Name` is a string or atom containing at least one character.

The `Name` given should correspond to the filename of the actual dynamically loadable object file residing in the directory given as `Path`, but *without* the extension (i.e. `.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 has been previously unloaded, but is still present due to open ports against it, a call to `load/2` will stop the unloading and keep the driver (as long as the `Path` is the same) and `ok` is returned. If one actually wants the object code to be reloaded, one uses *reload/2* or the low-level interface *try_load/3* instead. Please refer to the description of *different scenarios* for loading/unloading in the introduction.

If more than one process tries to load an already loaded driver withe the same `Path`, or if the same process tries to load it several times, the function will return `ok`. The emulator will keep track of the `load/2` calls, so that a corresponding number of `unload/2` calls will have to be done from the same process before the driver will actually get 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 several drivers with the same name but with different `Path` parameters.

> Note:
>
> Note especially that the `Path` is interpreted literally, so that all loaders of the same driver needs to give the same *literal*`Path` string, even though different paths might point out the same directory in the filesystem (due to 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 the *format_error/1* function.

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

The function throws a `badarg` exception if the parameters are not given as described above.

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

Types:

`Path = path()`

`Name = driver()`

`ErrorDesc = term()`

Works essentially as `load/2`, but will load the driver with other options. All ports that are using the driver will get killed with the reason `driver_unloaded` when the driver is to be unloaded.

The number of loads and unloads by different *users* influence the actual loading and unloading of a driver file. The port killing will therefore only happen when the *last user* unloads the driver, or 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 will give the same effect regarding the port killing.

The function throws a `badarg` exception if the parameters are not given as described above.

`monitor(Tag, Item) -> MonitorRef`

Types:

Tag = driver

Item = {Name, When}

Name = atom() | string()

When = loaded | unloaded | unloaded_only

MonitorRef = reference()

This function creates a driver monitor and works in many ways as the function *erlang:monitor/2*, does for processes. When a driver changes state, the monitor results in a monitor-message being sent to the calling process. The `MonitorRef` returned by this function is included in the message sent.

As with process monitors, each driver monitor set will only generate *one single message*. The monitor is "destroyed" after the message is sent and there is then no need to call *demonitor/1*.

The `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 given for consistence.

Item

The `Item` parameter specifies which driver one wants to monitor (the name of the driver) as well as which state change one wants to monitor. The parameter is a tuple of arity two whose first element is the driver name and second element is either of:

loaded

Notify me 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. One cannot monitor a future-to-be driver name for loading, that will only result in a `'DOWN'` message being immediately sent. Monitoring for loading is therefore most useful when triggered by the *try_load/3* function, where the monitor is created *because* the driver is in such a pending state.

Setting a driver monitor for `loading` will eventually lead 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 prior to creating a monitor for loading.

{'UP', reference(), driver, Name, permanent}

This message will be sent if reloading was expected, but the (old) driver made itself permanent prior to reloading. It will also be sent if the driver was permanent or statically linked in when trying to create the monitor.

{'DOWN', reference(), driver, Name, load_cancelled}

This message will arrive if reloading was underway, but the *user* having requested reload cancelled it by either 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 will arrive 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. Note that the translation has to be done in the same running erlang virtual machine as the error was detected in.

unloaded

Monitor when a driver gets unloaded. If one monitors a driver that is not present in the system, one will immediately get notified that the driver got unloaded. There is no guarantee that the driver was actually ever loaded.

A driver monitor for unload will eventually result in one of the following messages being sent:

{'DOWN', reference(), driver, Name, unloaded}

The driver instance monitored is now unloaded. As the unload might have been due to a `reload/2` request, the driver might once again have been loaded when this message arrives.

{'UP', reference(), driver, Name, unload_cancelled}

This message will be 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 when an `{ok, pending_driver}`) was returned from *try_unload/2*) for the last *user* of the driver and then a `{ok, already_loaded}` is returned from a call to *try_load/3*.

If one wants to *really* monitor when the driver gets unloaded, this message will distort the picture, no unloading was really done. The `unloaded_only` option creates a monitor similar to an `unloaded` monitor, but does never result in this message.

{'UP', reference(), driver, Name, permanent}

This message will be sent if unloading was expected, but the driver made itself permanent prior to unloading. It will also be 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` with the exception that the `{'UP', reference(), driver, Name, unload_cancelled}` message will never be sent, but the monitor instead persists until the driver *really* gets unloaded.

The function throws a `badarg` exception if the parameters are not given as described above.

`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 was 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 will return `{error, pending_process}`, but if there are no more users, the function call will hang until all open ports are closed.

> Note:
>
> Avoid mixing several *users* with driver reload requests.

If one wants to avoid hanging on open ports, one should use the *try_load/3* function instead.

The `Name` and `Path` parameters have exactly the same meaning as when calling the plain *load/2* function.

> Note:
>
> Avoid mixing several *users* with driver reload requests.

On success, the function returns `ok`. On failure, the function returns an opaque error, with the exception of the `pending_process` error described above. The opaque errors are to be translated into human readable form by the *format_error/1* function.

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

The function throws a `badarg` exception if the parameters are not given as described above.

`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 being killed when the last user disappears, the function wont hang waiting for ports to get closed.

For further details, see the *scenarios* in the module description and refer to the *reload/2* function description.

The function throws a `badarg` exception if the parameters are not given as described above.

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

Types:

Path = Name = string() | atom()

OptionList = [Option]

Option = {driver_options, DriverOptionList} | {monitor, MonitorOption} | {reload, ReloadOption}

DriverOptionList = [DriverOption]

DriverOption = kill_ports

MonitorOption = pending_driver | pending

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

This function provides more control than the `load/2`/`reload/2` and `load_driver/2`/`reload_driver/2` interfaces. It will never wait for completion of other operations related to the driver, but immediately return the status of the driver as either:

{ok, loaded}

The driver was actually loaded and is immediately usable.

{ok, already_loaded}

The driver was already loaded by another process and/or is in use by a living port. 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 due to the fact that an earlier instance of the driver is still waiting to get unloaded (there are open ports using it). Still, unload is expected when you are done with the driver. This return value will *mostly* happen when the `{reload,pending_driver}` or `{reload,pending}` options are used, but *can* happen when another *user* is unloading a driver in parallel and the `kill_ports` driver option is set. In other words, this return value will always need to be handled!

*{ok, pending_process}*or *{ok, pending_process, reference()}*

The load request is registered, but the loading is delayed due to the fact that 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 will *only* happen when the `{reload,pending}` option is used.

When the function returns `{ok, pending_driver}` or `{ok, pending_process}`, one might want to get information about when the driver is *actually* loaded. This can be achieved by using the `{monitor, PendingOption}` option.

When monitoring is requested, and a corresponding `{ok, pending_driver}` or `{ok, pending_process}` would be returned, the function will instead return a tuple `{ok, PendingStatus, reference()}` and the process will, at a later time when the driver actually gets loaded, get a monitor message. The monitor message one can expect is described in the *monitor/2* function description.

> ### Note:
>
> Note that in case of loading, monitoring can *not* only get triggered by using the `{reload, ReloadOption}` option, but also in special cases where the load-error is transient, why `{monitor, pending_driver}` should be used under basically *all* real world circumstances!

The function accepts the following parameters:

Path

The filesystem path to the directory where the driver object file is situated. The filename of the object file (minus extension) must correspond to the driver name (used in the name parameter) and the driver must identify itself with the very same name. The `Path` might be provided as an *io_list*, meaning it can be a list of other io_lists, characters (eight 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 should, by all *users*, be loaded using the same *literal*`Path`. The exception is when *reloading* is requested, in which case the `Path` may be specified differently. Note that all *users* trying to load the driver at a later time will need to use the *new*`Path` if the `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

The name parameter is the name of the driver to be used in subsequent calls to *open_port*. The name can be specified either as an `io_list()` or as an `atom()`. The name given when loading is used to find the actual object file (with the help of the `Path` and the system implied extension suffix, i.e. `.so`). The name by which the driver identifies itself must also be consistent with this `Name` parameter, much as a beam-file's module name much correspond to its filename.

OptionList

A number of options can be specified to control the loading operation. The options are given as a list of two-tuples, the tuples having the following values and meanings:

{driver_options, DriverOptionsList}

This option is to provide options that will change its general behavior and will "stick" to the driver throughout its lifespan.

The driver options for a given 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 actual name.

Currently the only allowed driver option is `kill_ports`, which means that all ports opened towards the driver are killed with the 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 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 will return a three-tuple `{ok, PendingStatus, reference()}`, where the `reference()` is the monitor ref for the driver monitor.

Only one `MonitorOption` can be specified and it is either the atom `pending`, which means that a monitor should be created whenever a load operation is delayed, and the atom `pending_driver`, in which a monitor is created whenever the operation is delayed due to open ports towards an otherwise unused driver. The `pending_driver` option is of little use, but is present for completeness, it is very well defined which reload-options might give rise to which delays. It might, however, be a good idea to use the same `MonitorOption` as the `ReloadOption` if present.

If reloading is not requested, it might still be useful to specify the `monitor` option, as forced unloads (`kill_ports` driver option or the `kill_ports` option to *try_unload/2*) will trigger a transient state where driver loading cannot be performed until all closing ports are actually closed. So, as `try_unload` can, in almost all situations, return `{ok, pending_driver}`, one should always specify at least `{monitor, pending_driver}` in production code (see the monitor discussion above).

{reload,RealoadOption}

This option is used when one wants to *reload* a driver from disk, most often in a code upgrade scenario. Having a `reload` option also implies that the `Path` parameter need *not* be consistent with earlier loads of the driver.

To reload a driver, the process needs to have previously loaded the driver, i.e there has to be an active *user* of the driver in the process.

The `reload` option can be either the atom `pending`, in which reloading is requested for any driver and will be effectuated when *all* ports opened against the driver are closed. The replacement of the driver will in this case take place regardless of if there are still pending *users* having the driver loaded! The option also triggers port-killing (if the `kill_ports` driver option is used) even though there are pending users, making it usable for forced driver replacement, but laying a lot of 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.

The more useful option is `pending_driver`, which means that reloading will be queued if the driver is *not* loaded by any other *users*, but the driver has opened ports, in which case `{ok, pending_driver}` will be returned (a `monitor` option is of course recommended).

If the driver is unloaded (not present in the system), the error code `not_loaded` will be returned. The `reload` option is intended for when the user has already loaded the driver in advance.

The function might return numerous errors, of which some only can be returned given a certain combination of options.

A number of errors are opaque and can only be interpreted by passing them to the *format_error/1* function, 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 has already been loaded with either other `DriverOptions` or a different *literal*`Path` argument.

This can happen even if a `reload` option is given, if the `DriverOptions` differ from the current.

{error, permanent}

The driver has requested itself to be permanent, making it behave like an erlang linked in driver and it can no longer be manipulated with this API.

{error, pending_process}

The driver is loaded by other *users* when the `{reload, pending_driver}` option was given.

{error, pending_reload}

Driver reload is already requested by another *user* when the `{reload, ReloadOption}` option was given.

{error, not_loaded_by_this_process}

Appears when the `reload` option is given. The driver `Name` is present in the system, but there is no *user* of it in this process.

{error, not_loaded}

Appears when the `reload` option is given. 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 the *format_error/1* function. Note that calls to `format_error` should be performed from the same running instance of the erlang virtual machine as the error was detected in, due to system dependent behavior concerning error values.

If the arguments or options are malformed, the function will throw a `badarg` exception.

`try_unload(Name, OptionList) -> {ok,Status} | {ok, PendingStatus, Ref} | {error, ErrorAtom}`

Types:

Name = string() | atom()

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, and it can trigger a monitor either due to other *users* still having the driver loaded or that there are open ports using the driver.

Unloading can be described as the process of telling the emulator that this particular part of the code in this particular process (i.e. this *user*) no longer needs the driver. That can, if there are no other users, trigger actual 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 the `kill_ports` option set, or if `kill_ports` was specified as an option to this function, all pending ports using this driver will get killed when unloading is done by the last *user*. If no port-killing is involved and there are open ports, the actual unloading is delayed until there are no more open ports using the driver. If, in this case, another *user* (or even this user) loads the driver again before the driver is actually unloaded, the unloading will never take place.

To allow the *user* that *requests unloading* to wait for *actual unloading* to take place, `monitor` triggers can be specified in much the same way as when loading. As *users* of this function however seldom are interested in more than decrementing the reference counts, monitoring is more seldom needed. If the `kill_ports` option is used however, monitor trigging is crucial, as the ports are not guaranteed to have been killed until the driver is unloaded, why a monitor should be triggered for at least the `pending_driver` case.

The possible monitor messages that can be expected are the same as when using the `unloaded` option to the *monitor/2* function.

The function will return 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 there are no more *users* requiring it to be loaded.

*{ok, pending_driver}*or *{ok, pending_driver, reference()}*

This return value 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 will actually be reloaded and the name and memory reclaimed.

This return value is valid even when the option `kill_ports` was used, as killing ports may not be a process that completes immediately. The condition is, in that case, however transient. Monitors are as always useful to detect when the driver is really unloaded.

*{ok, pending_process}*or *{ok, pending_process, reference()}*

The unload request is registered, but there are still other *users* holding the driver. Note that the term `pending_process` might refer to the running process, there might 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 actually the most common return value in the most common *scenario* described in the introduction.

The function accepts the following parameters:

Name

The name parameter is the name of the driver to be unloaded. The name can be specified either as an `io_list()` or as an `atom()`.

OptionList

The `OptionList` argument can be used to specify certain behavior regarding ports as well as triggering monitors under certain conditions:

kill_ports

Force killing of all ports opened using this driver, with the exit reason `driver_unloaded`, if you are the *lastuser* of the driver.

If there are other *users* having the driver loaded, this option will have no effect.

If one wants the consistent behavior of killing ports when the last *user* unloads, one should use the driver option `kill_ports` when loading the driver instead.

{monitor, MonitorOption}

This option creates a driver monitor if the condition given in `MonitorOptions` is true. The valid options are:

pending_driver

Create a driver monitor if the return value is to be `{ok, pending_driver}`.

pending

Create a monitor if the return value will be either `{ok, pending_driver}` or `{ok, pending_process}`.

The `pending_driverMonitorOption` is by far the most useful and it has to be used to ensure that the driver has really been unloaded and the ports closed whenever the `kill_ports` option is used or the driver may have been loaded with the `kill_ports` driver option.

By using the monitor-triggers in the call to `try_unload` one can be sure that the monitor is actually added before the unloading is executed, meaning that the monitor will always get properly triggered, which would not be the case if one called `erl_ddll:monitor/2` separately.

The function may return several error conditions, of which all are 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 has done no corresponding call to `try_load/3` if, and only if, there are *no users of the driver at all*, which may happen 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 given as described above.

`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 there are no more open ports using the driver, the driver will actually get unloaded. In all other cases, actual unloading will be delayed until all ports are closed and there are no remaining *users*.

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 usage scenarios, see the *description* in the beginning of this document.

The `ErrorDesc` returned is an opaque value to be passed further on to the *format_error/1* function. For more control over the operation, use the *try_unload/2* interface.

The function throws a `badarg` exception if the parameters are not given as described above.

`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 will get killed with the reason `driver_unloaded` and the driver will eventually get 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 usage scenarios, see the *description* in the beginning of this document.

The `ErrorDesc` returned is an opaque value to be passed further on to the *format_error/1* function. For more control over the operation, use the *try_unload/2* interface.

The function throws a `badarg` exception if the parameters are not given as described above.

`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.

More information about drivers can be obtained using one of the *info* functions.

`format_error(ErrorDesc) -> string()`

Types:

`ErrorDesc = term()`

Takes an `ErrorDesc` returned by load, unload or reload functions and returns a string which describes the error or warning.

> Note:
>
> Due to peculiarities in the dynamic loading interfaces on different platform, 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)!

SEE ALSO

erl_driver(4), 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

The error handler module defines what happens when certain types of errors occur.

Exports

undefined_function(Module, Function, Args) -> any()

Types:

Module = Function = atom()
Args = list()

A (possibly empty) list of arguments `Arg1,..,ArgN`

This function is evaluated if a call is made to `Module:Function(Arg1,.., ArgN)` and `Module:Function/N` is undefined. Note that `undefined_function/3` is evaluated inside the process making the original call.

If `Module` is interpreted, the interpreter is invoked and the return value of the interpreted `Function(Arg1,.., ArgN)` call is returned.

Otherwise, it returns, if possible, the value of `apply(Module, Function, Args)` after an attempt has been made to autoload `Module`. If this is not possible, the call to `Module:Function(Arg1,.., ArgN)` fails with exit reason `undef`.

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 has been made to autoload `Module`. If this is not possible, the call fails with exit reason `undef`.

Notes

The code in `error_handler` is complex and should not be changed without fully understanding the interaction between the error handler, the `init` process of the code server, and the I/O mechanism of the code.

Changes in the code which may seem small can cause a deadlock as unforeseen consequences may 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`. Error, warning 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 tty. Note 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` only has a primitive event handler, which buffers and prints the raw event messages. During system startup, the application Kernel replaces this with a *standard event handler*, by default one which writes nicely formatted output to tty. Kernel can also be configured so that events are logged to 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 tty. See *sasl(6)*.

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

Warning events was introduced in Erlang/OTP R9C. To retain backwards compatibility, these are by default tagged as errors, thus showing up as error reports in the logs. By using the command line flag `+W <w | i>`, they can instead be tagged as warnings or info. Tagging them as warnings may require rewriting existing user defined event handlers.

Data Types

```
report() = [{Tag :: term(), Data :: term()} | term()]
         | string()
         | term()
```

Exports

```
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`. The event is handled by the standard event handler.

```
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.

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.

```
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`.

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 (default), warnings or info, depending on the value of the command line flag `+W`.

```
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().
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
3>
User switch command
 --> q
os$ erl +W w
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
```

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`. The event is handled by the standard event handler. It is tagged either 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_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 either 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 either as an error, warning or info, depending on the value of *warning_map/0*.

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`. The event is handled by the standard event handler.

```
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.

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.

```
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`.

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* described below.

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)*.

tty(Flag) -> ok

Types:

Flag = boolean()

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

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

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 in combination with calling `tty(false)`, in order to have a silent system, where all standard events are logged to a file only. There can only be one active log file at a time.

`Request` is one of:

`{open, Filename}`

Opens the 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` could not be opened.

`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.

Events

All event handlers added to the error logger must handle the following events. `Gleader` is the group leader pid of the process which sent the event, and `Pid` is the process which 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, provided that warnings are set to be tagged as warnings.

`{warning_report, Gleader, {Pid, std_warning, Report}}`

Generated when `warning_report/1` is called, provided that warnings are set to be tagged as warnings.

`{warning_report, Gleader, {Pid, Type, Report}}`

Generated when `warning_report/2` is called, provided that 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.

Note that also a number of system internal events may be received, a catch-all clause last in the definition of the event handler callback function `Module:handle_event/2` is necessary. This also holds true for `Module:handle_info/2`, as there are a number of system internal messages the event handler must take care of as well.

SEE ALSO

gen_event(3), log_mf_h(3), kernel(6), sasl(6)

file

Erlang module

The module `file` provides an interface to the file system.

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

The Erlang VM supports file names in Unicode to a limited extent. Depending on how the VM is started (with the parameter `+fnu` or `+fnl`), file names given can contain characters > 255 and the VM system will convert file names back and forth to the native file name encoding.

The default behavior for Unicode character translation depends on to what extent the underlying OS/filesystem enforces consistent naming. On OSes where all file names are ensured to be in one or another encoding, Unicode is the default (currently this holds for Windows and MacOSX). On OSes with completely transparent file naming (i.e. all Unixes except MacOSX), ISO-latin-1 file naming is the default. The reason for the ISO-latin-1 default is that file names are not guaranteed to be possible to interpret according to the Unicode encoding expected (i.e. UTF-8), and file names that cannot be decoded will only be accessible by using "raw file names", in other word file names given as binaries.

As file names are traditionally not binaries in Erlang, applications that need to handle raw file names need to be converted, why the Unicode mode for file names is not default on systems having completely transparent file naming.

> Note:
> As of R14B01, the most basic file handling modules (`file`, `prim_file`, `filelib` and `filename`) accept raw file names, but the rest of OTP is not guaranteed to handle them, why Unicode file naming on systems where it is not default is still considered experimental.

Raw file names is a new feature in OTP R14B01, which allows the user to supply completely uninterpreted file names to the underlying OS/filesystem. They are supplied as binaries, where it is up to the user to supply a correct encoding for the environment. The function `file:native_name_encoding()` can be used to check what encoding the VM is working in. If the function returns `latin1` file names are not in any way converted to Unicode, if it is `utf8`, raw file names should be encoded as UTF-8 if they are to follow the convention of the VM (and usually the convention of the OS as well). Using raw file names is useful if you have a filesystem with inconsistent file naming, where some files are named in UTF-8 encoding while others are not. A file:list_dir on such mixed file name systems when the VM is in Unicode file name mode might return file names as raw binaries as they cannot be interpreted as Unicode file names. Raw file names can also be used to give UTF-8 encoded file names even though the VM is not started in Unicode file name translation mode.

Note that on Windows, `file:native_name_encoding()` returns `utf8` per default, which is the format for raw file names even on Windows, although the underlying OS specific code works in a limited version of little endian UTF16. As far as the Erlang programmer is concerned, Windows native Unicode format is UTF-8...

Data Types

bindings() = *erl_eval:binding_struct()*

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

fd() =

```
    #file_descriptor{module = undefined | module(),
                     data = undefined | term()}

filename() = string() | binary()

io_device() = pid() | fd()
```

As returned by *file:open/2*, a process handling IO protocols.

```
name() = 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 filesystem (i.e. 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 which is named from the POSIX error codes used in Unix, and in the runtime libraries of most C compilers.

```
date() =
    {Year :: integer() >= 1,
     Month :: integer() >= 1,
     Day :: integer() >= 1}

time() =
    {Hour :: integer() >= 0,
     Minute :: integer() >= 0,
     Second :: integer() >= 0}

date_time() = {date(), time()}
```

Must denote a valid date and time.

```
file_info() =
    #file_info{size = undefined | integer() >= 0,
               type = undefined
                    | device
                    | directory
                    | other
                    | regular
                    | symlink,
               access = undefined
                      | read
                      | write
                      | read_write
                      | none,
               atime = undefined | file:date_time(),
               mtime = undefined | file:date_time(),
               ctime = undefined | file:date_time(),
               mode = undefined | integer(),
               links = undefined | integer() >= 0,
               major_device = undefined | integer(),
               minor_device = undefined | integer(),
               inode = undefined | integer(),
               uid = undefined | integer(),
               gid = undefined | integer()}

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()}
```

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.

change_group(Filename, Gid) -> ok | {error, Reason}

Types:

```
Filename = name()
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()
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()*
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()*
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()*
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()*
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`, expect for some severe errors such as out of memory.

Note that if the option `delayed_write` was used when opening the file, `close/1` might 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()*
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. See *open/2* for a list of typical error codes.

`{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.

Example:

```
f.txt:  {person, "kalle", 25}.
        {person, "pelle", 30}.
```

```
1> file:consult("f.txt").
{ok,[{person,"kalle",25},{person,"pelle",30}]}
```

```
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()
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 e.g. `open/2`. `ByteCount` defaults to `infinity`, denoting an infinite number of bytes.

The 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 actually was copied, which may 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`.

del_dir(Dir) -> ok | {error, Reason}

Types:

Dir = *name()*

Reason = *posix()* | badarg

Tries to delete the directory `Dir`. The directory must be empty before it can be deleted. Returns `ok` if successful.

Typical error reasons are:

`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()*

Reason = *posix()* | badarg

Tries to delete the file `Filename`. Returns `ok` if successful.

Typical error reasons are:

`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 super-user.

`enotdir`

A component of the file name is not a directory. On some platforms, `enoent` is returned instead.

`einval`

`Filename` had an improper type, such as tuple.

> Warning:
>
> In a future release, a bad type for the `Filename` argument will probably generate an exception.

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

Types:

```
Filename = name()
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 actual 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. See `open/2` for a list of typical error codes.

`{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.

eval(Filename, Bindings) -> ok | {error, Reason}

Types:

```
Filename = name()
Bindings = bindings()
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. See *erl_eval(3)* about variable bindings.

file_info(Filename) -> {ok, FileInfo} | {error, Reason}

This function is obsolete. Use `read_file_info/1` instead.

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 may happen if read permission does not exist for the parent directories of the current directory.

Typical error reasons are:

`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`

`Drive` should be of the form "`Letter:`", for example "c:". Returns `{ok, Dir}` or `{error, Reason}`, where `Dir` is the current working directory of the drive specified.

This function returns `{error, enotsup}` on platforms which have no concept of current drive (Unix, for example).

Typical error reasons are:

`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()`
`Filenames = [filename()]`
`Reason = posix() | badarg`

Lists all the files in a directory. Returns `{ok, Filenames}` if successful. Otherwise, it returns `{error, Reason}`. `Filenames` is a list of the names of all the files in the directory. The names are not sorted.

Typical error reasons are:

`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()*
Reason = *posix()* | badarg

Tries to create the directory Dir. Missing parent directories are *not* created. Returns ok if successful.

Typical error reasons are:

eacces

Missing search or write permissions for the parent directories of Dir.

eexist

There is already a file or directory named Dir.

enoent

A component of Dir does not exist.

enospc

There is a no space 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()*
Reason = *posix()* | badarg

Makes a hard link from Existing to New, on platforms that support links (Unix). This function returns ok if the link was successfully created, or {error, Reason}. On platforms that do not support 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(Name1, Name2) -> ok | {error, Reason}

Types:

Name1 = Name2 = *name()*
Reason = *posix()* | badarg

This function creates a symbolic link `Name2` to the file or directory `Name1`, on platforms that support symbolic links (most Unix systems). `Name1` need not exist. This function returns `ok` if the link was successfully created, or `{error, Reason}`. On platforms that do not support symbolic links, `{error, enotsup}` is returned.

Typical error reasons:

`eacces`

Missing read or write permissions for the parent directories of `Name1` or `Name2`.

`eexist`

`Name2` already exists.

`enotsup`

Symbolic links are not supported on this platform.

native_name_encoding() -> latin1 | utf8

This function returns the configured default file name encoding to use for raw file names. Generally an application supplying file names raw (as binaries), should obey the character encoding returned by this function.

By default, the VM uses ISO-latin-1 file name encoding on filesystems and/or OSes that use completely transparent file naming. This includes all Unix versions except MacOSX, where the vfs layer enforces UTF-8 file naming. By giving the experimental option `+fnu` when starting Erlang, UTF-8 translation of file names can be turned on even for those systems. If Unicode file name translation is in effect, the system behaves as usual as long as file names conform to the encoding, but will return file names that are not properly encoded in UTF-8 as raw file names (i.e. binaries).

On Windows, this function also returns `utf8` by default. The OS uses a pure Unicode naming scheme and file names are always possible to interpret as valid Unicode. The fact that the underlying Windows OS actually encodes file names using little endian UTF-16 can be ignored by the Erlang programmer. Windows and MacOSX are the only operating systems where the VM operates in Unicode file name mode by default.

open(Filename, Modes) -> {ok, IoDevice} | {error, Reason}

Types:

Filename = *name()*
Modes = [*mode()*]
IoDevice = *io_device()*
Reason = *posix()* | badarg | system_limit

Opens the file `Filename` in the mode determined by `Modes`, which may contain one or more of the following items:

`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 if `write` is not combined with `read`, the file will be truncated.

`append`

The file will be opened for writing, and it will be created if it does not exist. Every write operation to a file opened with `append` will take place at the end of the file.

`exclusive`

The file, when opened for writing, is created if it does not exist. If the file exists, open will return `{error, eexist}`.

> Warning:
>
> This option does not guarantee exclusiveness on file systems that do not support 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 should be safe).

`raw`

The `raw` option allows faster access to a file, because 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, because they can only talk to an Erlang process. Instead, use the `read/2`, `read_line/1` and `write/2` functions.
- Especially if `read_line/1` is to be used on a `raw` file, it is recommended to combine this option with the `{read_ahead, Size}` option as line oriented I/O is inefficient without buffering.
- Only the Erlang process which 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`

When this option has been given, read operations on the file will return binaries rather than lists.

`{delayed_write, Size, Delay}`

If this option is used, the data in subsequent `write/2` calls is buffered until there are at least `Size` bytes 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, so the `write/2` calls should be for sizes significantly less than `Size`, and not interspersed by to many other file operations, for this to happen.

When this option is used, the result of `write/2` calls may prematurely be reported as successful, and if a write error should actually occur 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` might return `{error, enospc}` because there was not enough space on the disc for previously written data, and `close/1` should probably be called again since 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 KBytes, 2 seconds)

`{read_ahead, Size}`

This option activates read data buffering. If `read/2` calls are for significantly less than `Size` bytes, read operations towards 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 since the number of operating system calls is reduced.

The `read_ahead` buffer is also highly utilized by the `read_line/1` function in `raw` mode, why 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 KBytes)

`compressed`

Makes it possible to read or write gzip compressed files. The `compressed` option must be combined with either `read` or `write`, but not both. Note that the file size obtained with `read_file_info/1` will most 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. Note that the data supplied to file:write or returned by file:read still is byte oriented, this option only denotes how data is actually 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 (the file) module for reading and writing data, as the interfaces provided here work with byte-oriented data, while using other (Unicode) encodings makes the *io(3)* module's `get_chars`, `get_line` and `put_chars` functions 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 will be closed.

The allowed values for `Encoding` are:

`latin1`

The default encoding. Bytes supplied to i.e. file:write are written as is on the file, likewise bytes read from the file are returned to i.e. file:read as is. If the *io(3)* module is used for writing, the file can only cope with Unicode characters up to codepoint 255 (the ISO-latin-1 range).

`unicode` or `utf8`

Characters are translated to and from the UTF-8 encoding before being written to or read from the file. A file opened in this way might be readable using the file:read function, as long as no data stored on the file lies beyond the ISO-latin-1 range (0..255), but failure will occur if the data contains Unicode codepoints beyond that range. The file is best read with the functions in the Unicode aware *io(3)* module.

Bytes written to the file by any means are translated to UTF-8 encoding before actually 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 the *io:setopts/2* function, why a file can be analyzed in latin1 encoding for i.e. a BOM, positioned beyond the BOM and then be set for the right encoding before further reading.See the *unicode(3)* module for functions identifying BOM's.

This option is not allowed on `raw` files.

Returns:

`{ok, IoDevice}`

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

`{error, Reason}`

The file could not be opened.

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

> ### Note:
>
> In previous versions of `file`, modes were given as one of the atoms `read`, `write`, or `read_write` instead of a list. This is still allowed for reasons of backwards compatibility, but should not 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 may be a directory, a fifo, or a device.

`enotdir`

A component of the file name is not a directory. On some platforms, `enoent` is returned instead.

`enospc`

There is a 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()
Terms = [term()]
FullName = filename()
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 was successfully read. `FullName` is the full name of the file.

`{error, enoent}`

The file could not be found in any of the directories in `Path`.

`{error, atom()}`

An error occurred when opening the file or reading it. See *open/2* for a list of typical error codes.

`{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.

`path_eval(Path, Filename) -> {ok, FullName} | {error, Reason}`

Types:

```
Path = [Dir :: name()]
Filename = name()
FullName = filename()
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 file name, `Path` is ignored. Then reads and evaluates Erlang expressions, separated by '.' (or ',', a sequence of expressions is also an expression), from the file. The actual 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 was read and evaluated. `FullName` is the full name of the file.

`{error, enoent}`

The file could not be found in any of the directories in `Path`.

`{error, atom()}`

An error occurred when opening the file or reading it. See *open/2* for a list of typical error codes.

`{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.

`path_open(Path, Filename, Modes) ->`
`{ok, IoDevice, FullName} | {error, Reason}`

Types:

```
Path = [Dir :: name()]
Filename = name()
Modes = [mode()]
IoDevice = io_device()
FullName = filename()
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 file name, `Path` is ignored. Then opens the file in the mode determined by `Modes`. Returns one of the following:

`{ok, IoDevice, FullName}`

The file has been 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 could not be found in any of the directories in `Path`.

`{error, atom()}`

The file could not be opened.

```
path_script(Path, Filename) ->
               {ok, Value, FullName} | {error, Reason}
```

Types:

```
Path = [Dir :: name()]
Filename = name()
Value = term()
FullName = filename()
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 file name, `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 was read and evaluated. `FullName` is the full name of the file and `Value` the value of the last expression.

`{error, enoent}`

The file could not be found in any of the directories in `Path`.

`{error, atom()}`

An error occurred when opening the file or reading it. See *open/2* for a list of typical error codes.

`{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.

```
path_script(Path, Filename, Bindings) ->
```

```
              {ok, Value, FullName} | {error, Reason}
```

Types:

```
Path = [Dir :: name()]
Filename = name()
Bindings = bindings()
Value = term()
FullName = filename()
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()
Pid = pid()
```

If `Pid` is an IO device, that is, a pid returned from `open/2`, this function returns the filename, or rather:

`{ok, Filename}`

If this node's file server is not a slave, the file was opened by this node's file server, (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.

Note 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, i.e. 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).

Typical error reasons are:

`einval`

Either `Location` was illegal, or it evaluated to a negative offset in the file. Note 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 was beyond end of file.

As the position is given as a byte-offset, special caution has to be taken when working with files where `encoding` is set to something else than `latin1`, as not every byte position will be 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` has been opened in raw mode, some restrictions apply: `Location` is only allowed to be an integer; and the current position of the file is undefined after the operation.

As the position is given as a byte-offset, special caution has to be taken when working with files where `encoding` is set to something else than `latin1`, as not every byte position will be 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 that was 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, see *position/2* for details.

`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; and 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, see *position/2* for details.

`read(IoDevice, Number) -> {ok, Data} | eof | {error, Reason}`

Types:

`IoDevice = io_device() | atom()`
`Number = integer() >= 0`
`Data = string() | binary()`
`Reason = posix() | badarg | terminated`

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 might be represented by more than one byte on the file. The parameter `Number` always denotes the number of *characters* read from the file, why the position in the file might be moved a lot more than this number when reading a Unicode file.

Also if `encoding` is set to something else than `latin1`, the `read/3` call will fail if the data contains characters larger than 255, why the *io(3)* module 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 will be 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 was opened with another `encoding` than `latin1` and the data in the file can not be translated to the byte-oriented data that this function returns.

`read_file(Filename) -> {ok, Binary} | {error, Reason}`

Types:

`Filename = `*`name()`
`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 file name 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}`

Types:

`Filename = `*`name()`
`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 record `file_info` contains the following fields.

`size = integer()`

Size of file in bytes.

`type = device | directory | regular | other`

The type of the file.

`access = read | write | read_write | none`

The current system access to the file.

`atime = time()`

The last (local) time the file was read.

`mtime = time()`

The last (local) time the file was written.

`ctime = time()`

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()`

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()`

Number of links to the file (this will always be 1 for file systems which have no concept of links).

`major_device = integer()`

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()`

Only valid for character devices on Unix. In all other cases, this field is zero.

`inode = integer()`

Gives the `inode` number. On non-Unix file systems, this field will be zero.

`uid = integer()`

Indicates the owner of the file. Will be zero for non-Unix file systems.

`gid = integer()`

Gives the group that the owner of the file belongs to. Will be zero 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 file name 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`

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 a 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. It is however inefficient to use it on `raw` files if the file is not opened with the option `{read_ahead, Size}` specified, why 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 will fail if the data contains characters larger than 255, why the *io(3)* module 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 was 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 was opened with another `encoding` than `latin1` and the data on the file can not be translated to the byte-oriented data that this function returns.

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

Types:

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

This function 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 will be `{error,enotsup}`.

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}`

Types:

`Name = `*`name()`
`FileInfo = `*`file_info()`
`Reason = `*`posix()` **`| badarg`**

This function works like `read_file_info/1`, except that if `Name` is a symbolic link, information about the link will be returned in the `file_info` record and the `type` field of the record will be set to `symlink`.

If `Name` is not a symbolic link, this function returns exactly 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()*
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 file name 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 sub-directory 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()*
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 was read and evaluated. `Value` is the value of the last expression.

`{error, atom()}`

An error occurred when opening the file or reading it. See *open/2* for a list of typical error codes.

`{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.

`script(Filename, Bindings) -> {ok, Value} | {error, Reason}`

Types:

```
Filename = name()
Bindings = bindings()
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.

`set_cwd(Dir) -> ok | {error, Reason}`

Types:

```
Dir = name()
Reason = posix() | badarg
```

Sets the current working directory of the file server to `Dir`. Returns `ok` if successful.

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` had an improper type, such as tuple.

> Warning:
>
> In a future release, a bad type for the `Dir` argument will probably generate an exception.

sync(IoDevice) -> ok | {error, Reason}

Types:

IoDevice = *io_device()*
Reason = *posix()* | badarg | terminated

Makes sure 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.

Typical error reasons are:

`enospc`

Not enough space left to write the file.

datasync(IoDevice) -> ok | {error, Reason}

Types:

IoDevice = *io_device()*
Reason = *posix()* | badarg | terminated

Makes sure that any buffers kept by the operating system (not by the Erlang runtime system) are written to disk. In many ways it's resembles fsync but it not requires to update some of file's metadata such as the access time. On some platforms, this function might have no effect.

Applications that access databases or log files often write a tiny data fragment (e.g., one line in a log file) and then call fsync() immediately in order to ensure that the written data is physically stored on the harddisk. Unfortunately, fsync() will always initiate two write operations: one for the newly written data and another one in order 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 implementing the fdatasync syscall.

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 might result in several bytes actually being written to the file, as the byte range 0..255 might represent anything between one and four bytes depending on value and UTF encoding type.

Typical error reasons are:

`ebadf`

The file is not opened for writing.

`enospc`

There is a no space left on the device.

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

Types:

`Filename = name()`
`Bytes = iodata()`
`Reason = posix() | badarg | terminated | system_limit`

Writes the contents of the iodata term `Bytes` to the file `Filename`. The file is created if it does not exist. If it exists, the previous contents are overwritten. Returns `ok`, or `{error, Reason}`.

Typical error reasons are:

`enoent`

A component of the file name does not exist.

`enotdir`

A component of the file name is not a directory. On some platforms, `enoent` is returned instead.

`enospc`

There is a no space 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()`
`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 should not be used.

`write_file_info(Filename, FileInfo) -> ok | {error, Reason}`

Types:

`Filename = name()`
`FileInfo = file_info()`
`Reason = posix() | badarg`

Change 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 following fields are used from the record, if they are given.

`atime = time()`

The last (local) time the file was read.

`mtime = time()`

The last (local) time the file was written.

`ctime = time()`

On Unix, any value give for this field will be ignored (the "ctime" for the file will be set to the current time). On Windows, this field is the new creation time to set for the file.

`mode = integer()`

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()`

Indicates the owner of the file. Ignored for non-Unix file systems.

`gid = integer()`

Gives the group that the owner of the file belongs to. Ignored 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 file name 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` - IO error
- `eisdir` - illegal operation on a directory
- `eloop` - too many levels of symbolic links
- `emfile` - too many open files
- `emlink` - too many links
- `enametoolong` - file name 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, may block their calling thread for seconds. If this befalls 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 due to 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 several smaller, which enables the emulator so server other processes during the file operation. This gives 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 uses the file system of the node's host machine. 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 another node's, and the other node runs on some other host machine, they may have different file systems. This is seldom a problem, but you have now been warned.

A normal file is really a process so it can be used as an IO 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 another node's, also binaries are copied.

Caching data to reduce the number of file operations, or rather the number of calls to the file driver, will generally increase 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 `file: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 will run slightly faster, but consume more memory and cause more memory fragmentation. How much this affects your application is something that this simple benchmark can not predict.

If the size of each binary is increased to 64 bytes, it will also run slightly faster, but the code will be twice as clumsy. In the current implementation are binaries larger than 64 bytes 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 then with 64 bytes, and will cause much more memory fragmentation. Note 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. There is no need 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 `file:write(FD, [Bin1, Bin2 | Bin3])` will write 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 `file:open/2` makes the file driver cache data to reduce the number of operating system calls. The function `create_file/2` in the example above takes 60 seconds seconds without the `delayed_write` option, which is 2.6 times slower.

And, as a really bad example, `create_file_slow/2` above without the `raw`, `binary` and `delayed_write` options, that is it calls `file: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`.

Warnings

If an error occurs when accessing an open file with the `io` module, the process which handles the file will exit. The dead file process might hang if a process tries to access it later. This will be fixed in a future release.

SEE ALSO

filename(3)

gen_tcp

Erlang module

The `gen_tcp` module provides functions for communicating with sockets using the TCP/IP protocol.

The following code fragment provides 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),
    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 the *examples* section.

Data Types

`hostname() = inet:hostname()`

`ip_address() = inet:ip_address()`

Represents an address of a TCP socket. It is a tuple as explained in *inet(3)*.

`port_number() = 0..65535`

`posix() = inet:posix()`

See *inet(3); POSIX Error Codes*.

`socket()`

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

Exports

```
connect(Address, Port, Options) -> {ok, Socket} | {error, Reason}

connect(Address, Port, Options, Timeout) ->
           {ok, Socket} | {error, Reason}
```

Types:

```
Address = ip_address() | hostname()
Port = port_number()
Options = [Opt :: term()]
Timeout = timeout()
Socket = socket()
Reason = posix()
```

Connects to a server on TCP port `Port` on the host with IP address `Address`. The `Address` argument can be either a hostname, or an IP address.

The available options are:

`list`

Received `Packet` is delivered as a list.

`binary`

Received `Packet` is delivered as a binary.

`{ip, ip_address()}`

If the host has several network interfaces, this option specifies which one to use.

`{port, Port}`

Specify which local port number to use.

`{fd, int()}`

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

`inet6`

Set up the socket for IPv6.

`inet`

Set up the socket for IPv4.

`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 closed, the following message is delivered:

```
```

```
{tcp_closed, Socket}
```

If an error occurs on the socket, the following message is delivered:

```
{tcp_error, Socket, Reason}
```

unless `{active, false}` is specified in the option list for the socket, in which case packets are retrieved by calling `recv/2`.

The optional `Timeout` parameter specifies a timeout in milliseconds. The default value is `infinity`.

> Note:
>
> The default values for options given to `connect` can be affected by the Kernel configuration parameter `inet_default_connect_options`. See *inet(3)* for details.

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

Types:

`Port = port_number()`
`Options = [Opt :: term()]`
`ListenSocket = socket()`
`Reason = posix()`

Sets up a socket to listen on the 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 available options are:

`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 defaults to 5. The backlog value defines the maximum length that the queue of pending connections may grow to.

`{ip, ip_address()}`

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

`{fd, Fd}`

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

`inet6`

Set up the socket for IPv6.

`inet`

Set up the socket for IPv4.

`Opt`

See *inet:setopts/2*.

The returned socket `ListenSocket` can only be used in calls to `accept/1,2`.

> ### Note:
>
> The default values for options given to `listen` can be affected by the Kernel configuration parameter `inet_default_listen_options`. See *inet(3)* for details.

`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 | `*`posix()`

Accepts an incoming connection request on a listen socket. `Socket` must be a socket returned from `listen/2`. `Timeout` specifies a timeout value in ms, defaults to `infinity`.

Returns `{ok, Socket}` if a connection is established, or `{error, closed}` if `ListenSocket` is closed, or `{error, timeout}` if no connection is established within the specified time. May also return 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:

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

unless `{active, false}` was specified in the option list for the listen socket, in which case packets are retrieved by calling `recv/2`.

> ### Note:
>
> It is worth noting that 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.

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

Types:

Socket = *socket()*
Packet = string() | binary()
Reason = *posix()*

Sends a packet on a socket.

There is no `send` call with timeout option, you use the `send_timeout` socket option if timeouts are desired. See the *examples* section.

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 | *posix()*
HttpPacket = term()

See the description of `HttpPacket` in *erlang:decode_packet/3*.

This function receives a packet from a socket in passive mode. A closed socket is indicated by a return value `{error, closed}`.

The `Length` argument is only meaningful when the socket is in `raw` mode and denotes the number of bytes to read. If `Length` = 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 gets closed from the other side.

The optional `Timeout` parameter specifies a timeout in milliseconds. The default value is `infinity`.

controlling_process(Socket, Pid) -> ok | {error, Reason}

Types:

Socket = *socket()*
Pid = pid()
Reason = closed | not_owner | *posix()*

Assigns a new controlling process `Pid` to `Socket`. The controlling process is the process which receives messages from the socket. If called by any other process than the current controlling process, `{error, eperm}` is returned.

close(Socket) -> ok

Types:

Socket = *socket()*

Closes a TCP socket.

shutdown(Socket, How) -> ok | {error, Reason}

Types:

Socket = *socket()*
How = read | write | read_write
Reason = *posix()*

Immediately close a socket in one or two directions.

`How == write` means closing the socket for writing, reading from it is still possible.

To be able to handle that the peer has done a shutdown on the write side, the `{exit_on_close, false}` option is useful.

Examples

The following example illustrates usage of the {active,once} option and multiple accepts by implementing a server as a number of worker processes doing accept on one single listen socket. The start/2 function takes the number of worker processes as well as a port number to listen for incoming connections on. If `LPort` is specified as `0`, an ephemeral portnumber is used, why the start function returns the actual portnumber 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.
```

A simple client could look like this:

```
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 fact that the `send` call does not accept a timeout option, is because timeouts on send is handled through the socket option `send_timeout`. The behavior of a send operation with no receiver is in a very high degree defined by the underlying TCP stack, as well as the network infrastructure. If one wants to write code that handles a hanging receiver that might eventually cause the sender to hang on a `send` call, one writes code like the following.

Consider a process that receives data from a client process that is to be forwarded to a server on the network. The process has connected to the server via TCP/IP and does not get any acknowledge for each message it sends, but has to rely on the send timeout option to detect that the other end is unresponsive. We could use the `send_timeout` option 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, we can now detect send timeouts:

```
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 would suffice to detect timeouts on receive, as most protocols include some sort of acknowledgment from the server, but if the protocol is strictly one way, the `send_timeout` option comes in handy!

gen_udp

Erlang module

The `gen_udp` module provides functions for communicating with sockets using the UDP protocol.

Data Types

`hostname() = inet:hostname()`

`ip_address() = inet:ip_address()`

Represents an address of a TCP socket. It is a tuple as explained in *inet(3)*.

`port_number() = 0..65535`

`posix() = inet:posix()`

See *inet(3); POSIX Error Codes*.

`socket()`

As returned by open/1,2.

Exports

`open(Port) -> {ok, Socket} | {error, Reason}`

`open(Port, Opts) -> {ok, Socket} | {error, Reason}`

Types:

```
Port = port_number()
Opts = [Opt :: term()]
Socket = socket()
Reason = posix()
```

Associates a UDP port number (`Port`) with the calling process.

The available options are:

`list`

Received `Packet` is delivered as a list.

`binary`

Received `Packet` is delivered as a binary.

`{ip, ip_address()}`

If the host has several network interfaces, this option specifies which one to use.

`{fd, int()}`

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

`inet6`

Set up the socket for IPv6.

`inet`

Set up the socket for IPv4.

`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, they are delivered as messages:

```
{udp, Socket, IP, InPortNo, Packet}
```

Note that arriving UDP packets that are longer than the receive buffer option specifies, might be truncated without warning.

`IP` and `InPortNo` define the address from which `Packet` came. `Packet` is a list of bytes if the option `list` was specified. `Packet` is a binary if the option `binary` was 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.

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

Types:

`Socket = `*`socket()`
`Address = `*`ip_address()` **` | `*`hostname()`**
`Port = `*`port_number()`
`Packet = string() | binary()`
`Reason = not_owner | `*`posix()`

Sends a packet to the specified address and port. The `Address` argument can be either a hostname, or an IP address.

```
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 = `*`ip_address()`
`Port = `*`port_number()`
`Packet = string() | binary()`
`Reason = not_owner | `*`posix()`

This function receives a packet from a socket in passive mode.

The optional `Timeout` parameter specifies a timeout in milliseconds. The default value is `infinity`.

controlling_process(Socket, Pid) -> ok

Types:

Socket = *socket()*

Pid = pid()

Assigns a new controlling process `Pid` to `Socket`. The controlling process is the process which receives messages from the socket.

close(Socket) -> ok

Types:

Socket = *socket()*

Closes a UDP socket.

gen_sctp

Erlang module

The `gen_sctp` module provides functions for communicating with sockets using the SCTP protocol. The implementation assumes that the OS kernel supports SCTP **(RFC2960)** 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), and on 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, and later on 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.

Record definitions for the `gen_sctp` 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.

CONTENTS

- *DATA TYPES*
- *EXPORTS*
- *SCTP SOCKET OPTIONS*
- *SCTP EXAMPLES*
- *SEE ALSO*
- *AUTHORS*

Data Types

assoc_id()

An opaque term returned in for example #sctp_paddr_change{} that identifies an association for an SCTP socket. The term is opaque except for the special value `0` that has a meaning such as "the whole endpoint" or "all future associations".

hostname() = *inet:hostname()*

ip_address() = *inet:ip_address()*

Represents an address of an SCTP socket. It is a tuple as explained in *inet(3)*.

port_number() = 0..65535

posix() = *inet:posix()*

See *inet(3); POSIX Error Codes*.

```
sctp_option() = {mode, list | binary}
              | list
              | binary
```

```
| {active, true | false | once}
| {buffer, integer() >= 0}
| {tos, integer()}
| {priority, integer()}
| {dontroute, boolean()}
| {reuseaddr, boolean()}
| {linger, {boolean(), integer() >= 0}}
| {sndbuf, integer() >= 0}
| {recbuf, integer() >= 0}
| {sctp_rtoinfo, #sctp_rtoinfo{}}
| {sctp_associnfo, #sctp_assocparams{}}
| {sctp_initmsg, #sctp_initmsg{}}
| {sctp_autoclose, timeout()}
| {sctp_nodelay, boolean()}
| {sctp_disable_fragments, boolean()}
| {sctp_i_want_mapped_v4_addr, boolean()}
| {sctp_maxseg, integer() >= 0}
| {sctp_primary_addr, #sctp_prim{}}
| {sctp_set_peer_primary_addr,
   #sctp_setpeerprim{}}
| {sctp_adaptation_layer, #sctp_setadaptation{}}
| {sctp_peer_addr_params, #sctp_paddrparams{}}
| {sctp_default_send_param, #sctp_sndrcvinfo{}}
| {sctp_events, #sctp_event_subscribe{}}
| {sctp_delayed_ack_time, #sctp_assoc_value{}}
| {sctp_status, #sctp_status{}}
| {sctp_get_peer_addr_info, #sctp_paddrinfo{}}
```

One of the *SCTP Socket Options.*

`sctp_socket()`

Socket identifier returned from `open/*`.

Exports

`abort(Socket, Assoc) -> ok | {error, posix()}`

Types:

`Socket = sctp_socket()`

`Assoc = #sctp_assoc_change{}`

Abnormally terminates the association given by `Assoc`, without flushing of unsent data. The socket itself remains open. Other associations opened on this socket are still valid, and it can be used in new associations.

`close(Socket) -> ok | {error, posix()}`

Types:

`Socket = sctp_socket()`

Completely 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 timeout expires, the call returns and the data is flushed in the background.

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

Types:

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

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

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

Types:

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

Establishes a new association for the socket `Socket`, with the peer (SCTP server socket) given by `Addr` and `Port`. The `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` (i.e. source address and port) since the controlling process blocks until `connect/*` returns. *connect_init/** provides an alternative not subject to this limitation.

The result of `connect/*` is an `#sctp_assoc_change{}` event which contains, in particular, the new *Association ID*.

```
#sctp_assoc_change{
      state             = atom(),
      error             = atom(),
      outbound_streams  = int(),
      inbound_streams   = int(),
      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 got 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 actual stream numbers on the socket, which may be different from the requested values (`OutStreams` and `MaxInStreams` respectively) if the peer requires lower values.

The following values of `state` are possible:

- `comm_up`: association successfully established. This indicates a successful completion of `connect`.
- `cant_assoc`: association cannot be established (`connect/*` failure).

All other states do not normally occur in the output from `connect/*`. Rather, they may occur in `#sctp_assoc_change{}` events received instead of data in *recv/** calls. All of them indicate losing the association due to various error conditions, and are listed here for the sake of completeness. The `error` field may provide more detailed diagnostics.

- `comm_lost`;
- `restart`;
- `shutdown_comp`.

`connect_init(Socket, Addr, Port, Opts) -> ok | {error, posix()}`

Types:

`Socket = sctp_socket()`
`Addr = ip_address() | hostname()`
`Port = port_number()`
`Opts = [sctp_option()]`

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

`connect_init(Socket, Addr, Port, Opts, Timeout) ->`
`ok | {error, posix()}`

Types:

`Socket = sctp_socket()`
`Addr = ip_address() | hostname()`
`Port = port_number()`
`Opts = [sctp_option()]`
`Timeout = timeout()`

Initiates a new association for the socket `Socket`, with the peer (SCTP server socket) given 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 then the result of the association establishement 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/**, with the exception of the `Timeout` value.

The timer associated with `Timeout` only supervises IP resolution of `Addr`

`controlling_process(Socket, Pid) -> ok`

Types:

Socket = *sctp_socket()*
Pid = pid()

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 given by `Assoc`, with flushing of all unsent data. The socket itself remains open. Other associations opened on this socket are still valid, and it can be used in new associations.

listen(Socket, IsServer) -> ok | {error, Reason}

Types:

Socket = *sctp_socket()*
IsServer = boolean()
Reason = term()

Sets up a socket to listen on the IP address and port number it is bound to. `IsServer` must be `true` or `false`. In the contrast to TCP, in SCTP there is no listening queue length. If `IsServer` is `true` the socket accepts new associations, i.e. it will become an SCTP server socket.

open() -> {ok, Socket} | {error, *posix()*}

open(Port) -> {ok, Socket} | {error, *posix()*}

open(Opts) -> {ok, Socket} | {error, *posix()*}

open(Port, Opts) -> {ok, Socket} | {error, *posix()*}

Types:

Opts = [Opt]
Opt = {ip, IP} | {ifaddr, IP} | {port, Port} | *sctp_option()*
IP = *ip_address()* | any | loopback
Port = *port_number()*
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 one free port.

A default set of socket *options* is used. In particular, the socket is opened in *binary* and *passive* mode, and with reasonably large *kernel* and driver *buffers*.

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 = ip_address()
FromPort = port_number()
AncData = [#sctp_sndrcvinfo{}]
Data = binary()
     | string()
     | #sctp_sndrcvinfo{}
     | #sctp_assoc_change{}
     | #sctp_paddr_change{}
     | #sctp_adaptation_event{}
Reason = 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 timeout is `infinity`. `FromIP` and `FromPort` indicate the sender's address.

`AncData` is a list of Ancillary Data items which may 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, since such ancillary data provide an easy way of determining the association and stream over which the message has been received. (An alternative way would be to get the Association ID from the `FromIP` and `FromPort` using the *sctp_get_peer_addr_info* socket option, but this would still not produce the Stream number).

The actual `Data` received may be a `binary()`, or `list()` of bytes (integers in the range 0 through 255) depending on the socket mode, or an SCTP Event. The following SCTP Events are possible:

- *#sctp_sndrcvinfo{}*
- *#sctp_assoc_change{}*;
- ```
  #sctp_paddr_change{
        addr      = {ip_address(),port()},
        state     = atom(),
        error     = int(),
        assoc_id  = assoc_id()
  }
  ```

  Indicates change of the status of the peer's IP address given by `addr` within the 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 (e.g. `addr_unreachable`), the `error` field provides additional diagnostics. In such cases, the `#sctp_paddr_change{}` Event is automatically converted into an `error` term returned by `gen_sctp:recv`. The `error` field value can be converted into a string using `error_string/1`.

- ```
  #sctp_send_failed{
        flags     = true | false,
        error     = int(),
        info      = #sctp_sndrcvinfo{},
        assoc_id  = assoc_id()
        data      = binary()
  }
  ```

  The sender may receive this event if a send operation fails. The `flags` is a Boolean specifying whether the data have actually been transmitted over the wire; `error` provides extended diagnostics, use `error_string/1`; `info` is the original *#sctp_sndrcvinfo{}* record used in the failed *send/**, and `data` is 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 = int(),
        assoc_id       = assoc_id()
  }
  ```

  Delivered when a peer sends an Adaptation Layer Indication parameter (configured through the option *sctp_adaptation_layer*). Note 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 send/4 function 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 `Data` message over an existing association and given stream.

`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 and `undefined` for an unrecognized error.

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 those options which are explicitly listed below are allowed for SCTP sockets. Options can be set on the socket using `gen_sctp:open/1,2` or `inet:setopts/2`, retrieved using `inet:getopts/2`, and when calling `gen_sctp:connect/4,5` options can be changed.

`{mode, list|binary}`or just `list` or `binary`.

Determines the type of data returned from `gen_sctp:recv/1,2`.

`{active, true|false|once}`

- If `false` (passive mode, the default), the caller needs to do an explicit `gen_sctp:recv` call in order to retrieve the available data from the socket.
- If `true` (full active mode), the pending data or events are sent to the owning process.

  NB: 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, after that the mode is automatically re-set to passive. This provides flow control as well as the possibility for the receiver to listen for its incoming SCTP data interleaved with other inter-process messages.

`{buffer, int()}`

Determines the size of the user-level software buffer used by the SCTP driver. Not to be confused with `sndbuf` and `recbuf` options which correspond to the kernel socket buffers. It is recommended to have `val(buffer) >= max(val(sndbuf),val(recbuf))`. In fact, the `val(buffer)` is automatically set to the above maximum when `sndbuf` or `recbuf` values are set.

`{tos, int()}`

Sets the Type-Of-Service field on the IP datagrams being sent, to the given value, which effectively determines a prioritization policy for the outbound packets. The acceptable values are system-dependent. TODO: we do not provide symbolic names for these values yet.

`{priority, int()}`

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

`{dontroute, true|false}`

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

`{reuseaddr, true|false}`

By default `false`. If true, the local binding address `{IP,Port}` of the socket can be re-used immediately: no waiting in the CLOSE_WAIT state is performed (may be required for high-throughput servers).

`{linger, {true|false, int()}`

Determines the timeout in seconds for flushing unsent data in the `gen_sctp:close/1` socket call. If the 1st component of the value tuple is `false`, the 2nd one is ignored, which means that `gen_sctp:close/1` returns immediately not waiting for data to be flushed. Otherwise, the 2nd component is the flushing time-out in seconds.

`{sndbuf, int()}`

The size, in bytes, of the *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, int()}`

The size, in bytes, of the *kernel* recv 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).

`{sctp_rtoinfo, #sctp_rtoinfo{}}`

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

Determines re-transmission time-out parameters, in milliseconds, for the association(s) given by `assoc_id`. If `assoc_id = 0` (default) indicates the whole endpoint. See **RFC2960** and **Sockets API Extensions for SCTP** for the exact semantics of the fields values.

`{sctp_associnfo, #sctp_assocparams{}}`

```
#sctp_assocparams{
      assoc_id                = assoc_id(),
      asocmaxrxt              = int(),
      number_peer_destinations = int(),
      peer_rwnd               = int(),
      local_rwnd              = int(),
      cookie_life             = int()
}
```

Determines association parameters for the association(s) given 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   = int(),
     max_instreams  = int(),
     max_attempts   = int(),
     max_init_timeo = int()
}
```

Determines the default parameters which this socket attempts to negotiate with its peer while establishing an association with it. Should 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`: max number of in-bound streams;
- `max_attempts`: max re-transmissions while establishing an association;
- `max_init_timeo`: time-out in milliseconds for establishing an association.

`{sctp_autoclose, int()|infinity}`

Determines the time (in seconds) after which an idle association is automatically closed.

`{sctp_nodelay, true|false}`

Turns on|off the Nagle algorithm for merging small packets into larger ones (which improves throughput at the expense of latency).

`{sctp_disable_fragments, true|false}`

If `true`, induces an error on an attempt to send a message which is larger than the current PMTU size (which would require fragmentation/re-assembling). Note 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, int()}`

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 given by `assoc_id`, `{IP,Port}` must be one of the peer's addresses. This option determines that the given address is treated by the local SCTP stack as the peer's primary address.

`{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 that it should use `{IP, Port}` as the primary address of the local endpoint for the association given by `assoc_id`.

`{sctp_adaptation_layer, #sctp_setadaptation{}}`

```
#sctp_setadaptation{
      adaptation_ind = int()
}
```

When set, requests that the local endpoint uses the value given by `adaptation_ind` as the Adaptation Indication parameter for establishing new associations. See **RFC2960** and **Sockets API Extenstions for SCTP** for more details.

`{sctp_peer_addr_params, #sctp_paddrparams{}}`

```
#sctp_paddrparams{
      assoc_id   = assoc_id(),
      address    = {IP, Port},
      hbinterval = int(),
      pathmaxrxt = int(),
      pathmtu    = int(),
      sackdelay  = int(),
      flags      = list()
}
IP = ip_address()
Port = port_number()
```

This option determines various per-address parameters for the association given by `assoc_id` and the peer address `address` (the SCTP protocol supports multi-homing, so more than 1 address can correspond to a given association).

- `hbinterval`: heartbeat interval, in milliseconds;
- `pathmaxrxt`: max 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`: enable heartbeat;
  - `hb_disable`: disable heartbeat;
  - `hb_demand`: initiate heartbeat immediately;
  - `pmtud_enable`: enable automatic Path MTU discovery;
  - `pmtud_disable`: disable automatic Path MTU discovery;
  - `sackdelay_enable`: enable SAC delay;
  - `sackdelay_disable`: disable SAC delay.

`{sctp_default_send_param, #sctp_sndrcvinfo{}}`

```
#sctp_sndrcvinfo{
      stream     = int(),
      ssn        = int(),
      flags      = list(),
      ppid       = int(),
      context    = int(),
```

```
      timetolive = int(),
      tsn        = int(),
      cumtsn     = int(),
      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 a default values for subsequent `gen_sctp:send`calls on the association given by `assoc_id`. `assoc_id = 0` (default) indicates the whole endpoint. The following fields typically need to 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 `gen_sctp:send` overwrites the primary peer address;
  - `abort`: abort the current association without flushing any unsent data;
  - `eof`: gracefully shut down the current association, with flushing of unsent data.

Other fields are rarely used. See **RFC2960** and **Sockets API Extensions for SCTP** for full information.

`{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 (via *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 = int()
}
```

Rarely used. Determines the ACK time (given by `assoc_value` in milliseconds) for the given association or the whole endpoint if `assoc_value = 0` (default).

`{sctp_status, #sctp_status{}}`

```
#sctp_status{
      assoc_id            = assoc_id(),
      state               = atom(),
```

```
      rwnd                = int(),
      unackdata           = int(),
      penddata            = int(),
      instrms             = int(),
      outstrms            = int(),
      fragmentation_point = int(),
      primary             = #sctp_paddrinfo{}
}
```

This option is read-only. It determines the status of the SCTP association given by `assoc_id`. Possible values of `state` follows. The state designations are mostly self-explanatory. `state_empty` is the default which means that no other state is active:

- `sctp_state_empty`
- `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`

The semantics of other fields is the following:

- `sstat_rwnd`: the association peer's current receiver window size;
- `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 will occur;
- `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,
      cwnd      = int(),
      srtt      = int(),
      rto       = int(),
      mtu       = int()
}
IP = ip_address()
Port = port_number()
```

This option is read-only. It determines the parameters specific to the peer's address given by `address` within the association given by `assoc_id`. The `address` field must be set by the caller; all other fields are filled in on return. If `assoc_id = 0` (default), the `address` is automatically translated into the corresponding association ID. This option is rarely used; see **RFC2960** and **Sockets API Extensions for SCTP** for the semantics of all fields.

SCTP EXAMPLES

- Example of an Erlang SCTP Server which 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([{ip,IP},{port,Port}],[{recbuf,65536}]),
    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 which interacts with the above Server. Note that in this example, the Client creates an association with the Server with 5 outbound streams. For this reason, 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 the corresponding Event being 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 very simple Erlang SCTP Client which 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

inet(3), *gen_tcp(3)*, *gen_udp(3)*, **RFC2960** (Stream Control Transmission Protocol), **Sockets API Extensions for SCTP.**

global

Erlang module

This documentation describes the Global module which consists of the following functionalities:

- registration of global names;
- global locks;
- maintenance of the fully connected network.

These services are controlled via the process `global_name_server` which exists on every node. The global name server is started automatically when a node is started. With the term *global* is meant over a system consisting of several 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 implemented, 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 will also be 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. When any action in taken which results 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 instance, the specified resource could be a pid. When a global lock is set, access to the locked resource is denied for all other resources other than the lock requester.

Both the registration and lock functionalities are atomic. All nodes involved in these actions will 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 which runs a globally registered process goes down, the name will be globally unregistered. To this end the global name server subscribes to `nodeup` and `nodedown` messages sent from the `net_kernel` module. Relevant Kernel application variables in this context are `net_setuptime`, `net_ticktime`, and `dist_auto_connect`. See also *kernel(6)*.

The name server will also maintain 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` will make sure that also `N1` and `N3` are connected. If this is not desired, the command line flag `-connect_all false` can be used (see also *erl(1)*). In this case the name registration facility cannot be used, but the lock mechanism will still work.

If the global name server fails to connect nodes (`N1` and `N3` in the example above) a warning event is sent to the error logger. The presence of such an event does not exclude the possibility that the nodes will later connect--one can for example try the command `rpc:call(N1, net_adm, ping, [N2])` in the Erlang shell--but it indicates some kind of problem with the network.

> ### Note:
>
> If the fully connected network is not set up properly, the first thing to try is 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()

This function can be used as a name resolving function for `register_name/3` and `re_register_name/3`. It unregisters both pids, and sends the message `{global_name_conflict, Name, OtherPid}` to both processes.

random_exit_name(Name, Pid1, Pid2) -> none

Types:

Name = term()

Pid1 = Pid2 = pid()

This function can be used as a name resolving function for `register_name/3` and `re_register_name/3`. It randomly chooses one of the pids for registration and kills the other one.

random_notify_name(Name, Pid1, Pid2) -> none

Types:

Name = term()

Pid1 = Pid2 = pid()

This function can be used as a name resolving function for `register_name/3` and `re_register_name/3`. It randomly chooses one of the pids for registration, and sends the message `{global_name_conflict, Name}` to the other pid.

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

Globally associates the 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, the Resolve function 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.

There are three pre-defined resolve functions: 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 will be selected as correct while the other is killed.

This function is completely synchronous. This means that 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 OTP R10 did not check if the process was already registered. As a consequence the global name table could 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 lists of all globally registered names.

```
re_register_name(Name, Pid) -> term()

re_register_name(Name, Pid, Resolve) -> term()
```

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.

The Resolve function has the same behavior as in register_name/2,3.

send(Name, Msg) -> Pid

Types:

Name = Msg = term()
Pid = pid()

Sends the message Msg to the pid globally registered as Name.

Failure: If Name is not a globally registered name, the calling function will exit 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 will try 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 given, infinity is used.

This function is completely synchronous.

If a process which 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. But if some processes try to lock two or more resources, a deadlock may occur. It is up to the application to detect and rectify a deadlock.

> Note:
>
> Some values of ResourceId should be avoided or Erlang/OTP will not work properly. A list of resources to avoid: global, dist_ac, mnesia_table_lock, mnesia_adjust_log_writes, pg2.

sync() -> ok | {error, Reason :: term()}

Synchronizes the global name server with all nodes known to this node. These are the nodes which are returned from `erlang:nodes()`. When this function returns, the global name server will receive 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 failed. If `Retries` is set to `infinity`, the transaction will not abort.

`infinity` is the default setting and will be used if no value is given 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

The global group function makes it possible to group the nodes in a system into partitions, each partition having its own global name space, refer to `global(3)`. These partitions are called global groups.

The main advantage of dividing systems to 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)*, *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, or the behavior of the system is undefined.
- *All* nodes in the system should belong to exactly one global group.

In the following description, 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 == normal`.

```
group_name() = atom()
```

```
publish_type() = hidden | normal
```

A node started with the command line flag `-hidden`, see *erl(1)*, is said to be a *hidden* node. A hidden node will establish 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 == hidden`, is said to be a hidden global group. All nodes in a hidden global group are hidden nodes, regardless if they are started with the `-hidden` command line flag 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 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 element of the list is a tuple. The order of the tuples is not defined.

`{state, State}`

If the local node is part of a global group, `State == synced`. If no global groups are defined, `State == 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 which have subscribed to `nodeup` and `nodedown` messages.

monitor_nodes(Flag) -> ok

Types:

Flag = boolean()

Depending on `Flag`, the calling process starts subscribing (`Flag == true`) or stops subscribing (`Flag == false`) to node status change messages.

A process which has subscribed will receive 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 which 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 the `Where` argument is not provided -- in any global group. The global groups are searched in the order in which they appear in the value of the `global_groups` configuration parameter.

If `Name` is found, the 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 check 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)`).

Failure: `{error, {'invalid global_groups definition', Bad}}` if the `global_groups` configuration parameter 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 the `Where` argument is not provided -- in any global group. The global groups are searched in the order in which they appear in the value of the `global_groups` configuration parameter.

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

NOTE

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 may produce an incorrect or misleading result. For example, the isolated node may not have accurate information about registered names in its global group.

Note also that the `send/2,3` function is not secure.

Distribution of applications is highly dependent of the global group definitions. It is not recommended that an application is distributed over several global groups of the obvious reason that the registered names may be moved to another global group at failover/takeover. There is nothing preventing doing this, but the application code must in such case handle the situation.

SEE ALSO

erl(1), *global(3)*

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 (default is 60 seconds), the system can be rebooted. Also, if the system is equipped with a hardware watchdog timer and is running Solaris, the watchdog can be used to supervise the entire system.

An Erlang runtime system to be monitored by a heart program, should be started with the command line flag `-heart` (see also *erl(1)*. The `heart` process is then started automatically:

```
% erl -heart ...
```

If the system should be rebooted because of missing heart-beats, or a terminated Erlang runtime system, the environment variable `HEART_COMMAND` has to be set before the system is started. If this variable is not set, a warning text will be printed but the system will not reboot. However, if the hardware watchdog is used, it will trigger a reboot `HEART_BEAT_BOOT_DELAY` seconds later nevertheless (default is 60).

To reboot on the WINDOWS platform `HEART_COMMAND` can be set to `heart -shutdown` (included in the Erlang delivery) or of course to any other suitable program which can activate a reboot.

The hardware watchdog will not be started under Solaris if the environment variable `HW_WD_DISABLE` is set.

The `HEART_BEAT_TIMEOUT` and `HEART_BEAT_BOOT_DELAY` environment variables can be used to configure the heart timeouts, they 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.

It should be noted that if the system clock is adjusted with more than `HEART_BEAT_TIMEOUT` seconds, `heart` will timeout and try to reboot the system. This can happen, for example, if the system clock is adjusted automatically by use of NTP (Network Time Protocol).

In the following descriptions, all function fails with reason `badarg` if `heart` is not started.

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 should be used in order to reboot the system. The new Erlang runtime system will (if it misbehaves) use the environment variable `HEART_COMMAND` to reboot.

Limitations: The length of the `Cmd` command string must be less than 2047 characters.

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()

Get the temporary reboot command. If the command is cleared, the empty string will be returned.

inet

Erlang module

Provides access to TCP/IP protocols.

See also *ERTS User's Guide*, *Inet configuration* for more information on how to configure an Erlang runtime system for IP communication.

Two Kernel configuration parameters affect the behaviour 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`, and `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 listensocket options are inherited, why no such application variable is needed for `accept`.

Using the Kernel configuration parameters mentioned above, one can set default options for all TCP sockets on a node. This should be used with care, but options like `{delay_send,true}` might be specified in this way. An example of starting an Erlang node with all sockets using delayed send could look like this:

```
$ erl -sname test -kernel \
inet_default_connect_options '[{delay_send,true}]' \
inet_default_listen_options '[{delay_send,true}]'
```

Note that the default option `{active, true}` currently cannot be changed, for internal reasons.

Addresses as inputs to functions can be either a string or a tuple. For instance, the IP address 150.236.20.73 can be passed to `gethostbyaddr/1` either as the string "150.236.20.73" or as the 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
                 {16#FFFF,0,0,0,0,0,(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}
```

A function that may be useful is `inet_parse:address/1`:

```
1> inet_parse:address("192.168.42.2").
{ok,{192,168,42,2}}
```

```
2> inet_parse:address("FFFF::192.168.42.2").
{ok,{65535,0,0,0,0,0,49320,10754}}
```

Data Types

```
hostent() =
    #hostent{h_name = undefined | inet:hostname(),
             h_aliases = [inet:hostname()],
             h_addrtype = undefined | inet | inet6,
             h_length = undefined | 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}

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

An atom which is named from the Posix error codes used in Unix, and in the runtime libraries of most C compilers. See *POSIX Error Codes*.

```
socket()
```

See *gen_tcp(3)* and *gen_udp(3)*.

```
family_option() = inet | inet6
```

Exports

```
close(Socket) -> ok
```

Types:

```
Socket = socket()
```

Closes a socket of any type.

get_rc() -> [{Par :: any(), Val :: any()}]

Returns the state of the Inet configuration database in form of a list of recorded configuration parameters. (See the ERTS User's Guide, Inet configuration, for more information). Only parameters with other than default values are returned.

format_error(Posix) -> string()

Types:

Posix = *posix()*

Returns a diagnostic error string. See the section below for possible `Posix` values and the corresponding strings.

getaddr(Host, Family) -> {ok, Address} | {error, *posix()*}

Types:

Host = *ip_address()* | *hostname()*
Family = *family_option()*
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 = *family_option()*
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 given an address.

gethostbyname(Hostname) -> {ok, Hostent} | {error, *posix()*}

Types:

Hostname = *hostname()*
Hostent = *hostent()*

Returns a `hostent` record given a hostname.

gethostbyname(Hostname, Family) ->
{ok, Hostent} | {error, *posix()*}

Types:

```
Hostname = hostname()
Family = family_option()
Hostent = hostent()
```

Returns a `hostent` record given a hostname, restricted to the given address family.

```
gethostname() -> {ok, Hostname}
```

Types:

```
Hostname = string()
```

Returns the local hostname. Will never fail.

```
getifaddrs() -> {ok, Iflist} | {error, posix()}
```

Types:

```
Iflist = [{Ifname, [Ifopt]}]
Ifname = string()
Ifopt = {flag, [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's addresses. `Ifname` is a Unicode string. `Hwaddr` is hardware dependent, e.g on Ethernet interfaces it is the 6-byte Ethernet address (MAC address (EUI-48 address)).

The `{addr,Addr}`, `{netmask,_}` and `{broadaddr,_}` tuples are repeated in the result list iff the interface has multiple addresses. If you come across an interface that has multiple `{flag,_}` or `{hwaddr,_}` tuples you have a really strange interface or possibly a bug in this function. The `{flag,_}` tuple is mandatory, all other optional.

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

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

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

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

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

Types:

Socket = term()

Options = [Opt | RawOptReq]

Opt = atom()

RawOptReq = {raw, Protocol, OptionNum, ValueSpec}

Protocol = integer()

OptionNum = integer()

ValueSpec = ValueSize | ValueBin

ValueSize = integer()

ValueBin = binary()

OptionValues = [{Opt, Val} | {raw, Protocol, OptionNum, ValueBin}]

Gets one or more options for a socket. See *setopts/2* for a list of available options.

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 simply left out in the returned list. An error tuple is only returned when getting options for the socket is impossible (i.e. the socket is closed or the buffer size in a raw request is too large). This behavior is kept for backward compatibility reasons.

A `RawOptReq` 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.

The `RawOptReq` consists of the 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 should be used when the underlying `getsockopt` requires *input* in the argument field, in which case the size of the binary should 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.

As an example, consider a Linux machine where the `TCP_INFO` option could be used to collect TCP statistics for a socket. Lets say we're interested in the `tcpi_sacked` field of the `struct tcp_info` filled in when asking for `TCP_INFO`. To be able to access this information, we need to know both the numeric value of the protocol level `IPPROTO_TCP`, the numeric value of the option `TCP_INFO`, the size of the `struct tcp_info` and the size and offset of the specific field. By inspecting the headers or writing a small C program, we found `IPPROTO_TCP` to be 6, `TCP_INFO` to be 11, the structure size to be 92 (bytes), the offset of `tcpi_sacked` to be 28 bytes and the actual value to be a 32 bit integer. We could use the following code 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 OS and the kernel version prior to executing anything similar to the code above.

getstat(Socket)

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

Types:

Socket = term()

Options = [Opt]

OptionValues = [{Opt, Val}]

Opt, Val -- see below

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, sen`

The following options are available:

`recv_avg`

Average size of packets in bytes received to the socket.

`recv_cnt`

Number of packets received to the socket.

`recv_dvi`

Average packet size deviation in bytes received to the socket.

`recv_max`

The size of the largest packet in bytes received to the socket.

`recv_oct`

Number of bytes received to 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 received sent from the socket.

`send_max`

The size of the largest packet in bytes sent from the socket.

`send_oct`

Number of bytes sent from the socket.

peername(Socket) -> {ok, {Address, Port}} | {error, *posix()*}

Types:

Socket = *socket()*

Address = *ip_address()*

Port = integer() >= 0

Returns the address and port for the other end of a connection.

`port(Socket) -> {ok, Port} | {error, any()}`

Types:

Socket = socket()

Port = integer()

Returns the local port number for a socket.

`sockname(Socket) -> {ok, {Address, Port}} | {error, posix()}`

Types:

`Socket = socket()`

`Address = ip_address()`

`Port = integer() >= 0`

Returns the local address and port number for a socket.

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

Types:

Socket = term()

Options = [{Opt, Val} | {raw, Protocol, Option, ValueBin}]

Protocol = integer()

OptionNum = integer()

ValueBin = binary()

Opt, Val -- see below

Sets one or more options for a socket. The following options are available:

`{active, true | false | once}`

If the value is `true`, which is the default, everything received from the socket will be 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` or `gen_udp:recv/2,3` (depending on the type of socket).

If the value is `once` (`{active, once}`), *one* data message from the socket will be sent to the process. To receive one more message, `setopts/2` must be called again with the `{active, once}` option.

When using `{active, once}`, the socket changes behaviour automatically when data is received. This can sometimes be confusing in combination with connection oriented sockets (i.e. `gen_tcp`) as a socket with `{active, false}` behaviour reports closing differently than a socket with `{active, true}` behaviour. To make programming easier, a socket where the peer closed and this was detected while in `{active, false}` mode, will still generate the message `{tcp_closed,Socket}` when set to `{active, once}` or `{active, true}` mode. It is therefore safe to assume that the message `{tcp_closed,Socket}`, possibly followed by socket port termination (depending on the `exit_on_close` option) will eventually appear when a socket changes back and forth between `{active, true}` and `{active, false}` mode. However, *when* peer closing is detected is all up to the underlying TCP/IP stack and protocol.

Note that `{active,true}` mode provides no flow control; a fast sender could easily overflow the receiver with incoming messages. Use active mode only if your high-level protocol provides its own flow control (for instance, acknowledging received messages) or the amount of data exchanged is small. `{active,false}` mode or use of the `{active, once}` mode provides flow control; the other side will not be able send faster than the receiver can read.

`{broadcast, Boolean}`(UDP sockets)

Enable/disable permission to send broadcasts.

`{delay_send, Boolean}`

Normally, when an Erlang process sends to a socket, the driver will try to immediately send the data. If that fails, the driver will use any means available to queue up the message to be sent whenever the operating system says it can handle it. Setting `{delay_send, true}` will make *all* messages queue up. This makes the messages actually sent onto the network be larger but fewer. The option actually affects the scheduling of send requests versus Erlang processes instead of changing any real property of the socket. Needless to say it is an implementation specific option. Default is `false`.

`{dontroute, Boolean}`

Enable/disable routing bypass for outgoing messages.

`{exit_on_close, Boolean}`

By default this option is set to `true`.

The only reason to set it to `false` is if you want to continue sending data to the socket after a close has been detected, for instance if the peer has used *gen_tcp:shutdown/2* to shutdown the write side.

`{header, Size}`

This option is only meaningful if the `binary` option was specified when the socket was created. If the `header` option is specified, the first `Size` number bytes of data received from the socket will be elements of a list, and the rest of the data will be a binary given as the tail of the same list. If for example `Size == 2`, the data received will match `[Byte1,Byte2|Binary]`.

`{keepalive, Boolean}`(TCP/IP sockets)

Enables/disables periodic transmission on a connected socket, when no other data is being exchanged. If the other end does not respond, the connection is considered broken and an error message will be sent to the controlling process. Default disabled.

`{nodelay, Boolean}`(TCP/IP sockets)

If `Boolean == true`, the `TCP_NODELAY` option is turned on for the socket, which means that even small amounts of data will be sent immediately.

`{packet, PacketType}`(TCP/IP sockets)

Defines the type of packets to use for a socket. The following values are valid:

`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 length of header can be one, two, or four bytes; containing an unsigned integer in big-endian byte order. Each send operation will generate the header, and the header will be stripped off on each receive operation.

In current implementation 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, there will be one message 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*. A socket in passive mode will return `{ok, HttpPacket}` from `gen_tcp:recv` while an active socket will send messages like `{http, Socket, HttpPacket}`.

Note that the packet type `httph` is not needed when reading from a socket.

`{packet_size, Integer}`(TCP/IP sockets)

Sets the max allowed length of the packet body. If the packet header indicates that the length of the packet is longer than the max allowed length, the packet is considered invalid. The same happens if the packet header is too big for the socket receive buffer.

`{read_packets, Integer}`(UDP sockets)

Sets the max 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. The default is 5, and if this parameter is set too high the system can become unresponsive due to UDP packet flooding.

`{recbuf, Integer}`

Gives the size of the receive buffer to use for the socket.

`{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 will return `{error,timeout}`. How much of a packet that actually got sent is unknown, why the socket should be closed whenever a timeout has occurred (see `send_timeout_close`). Default is `infinity`.

`{send_timeout_close, Boolean}`

Only allowed for connection oriented sockets.

Used together with `send_timeout` to specify whether the socket will be automatically closed when the send operation returns `{error,timeout}`. The recommended setting is `true` which will automatically close the socket. Default is `false` due to backward compatibility.

`{sndbuf, Integer}`

Gives the size of the send buffer to use for the socket.

`{priority, Integer}`

Sets the SO_PRIORITY socket level option on platforms where this is implemented. The behaviour and allowed range varies on different systems. The option is ignored on platforms where the option is not implemented. Use with caution.

`{tos, Integer}`

Sets IP_TOS IP level options on platforms where this is implemented. The behaviour and allowed range varies on different systems. The option is ignored on platforms where the option is not implemented. Use with caution.

In addition to the options mentioned above, *raw* option specifications can be used. The raw options are specified as a tuple of arity four, beginning with the tag `raw`, followed by the protocol level, the option number and the actual option value specified as a binary. This corresponds to the second, third and fourth argument to the `setsockopt` call in the C socket API. The option value needs to be coded in the native endianess of the platform and, if a structure is required, needs to follow the struct alignment conventions on the specific platform.

Using raw socket options require detailed knowledge about the current operating system and TCP stack.

As an example of the usage of raw options, consider a Linux system where you want to set the `TCP_LINGER2` option on the `IPPROTO_TCP` protocol level in the stack. You know that on this particular system it defaults to 60 (seconds), but you would like to lower it to 30 for a particular socket. The `TCP_LINGER2` option is not explicitly supported by inet, but you know that the protocol level translates to the number 6, the option number to the number 8 and the value is to be given as a 32 bit integer. You can use this line of code 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 given out of range, it could be a good idea to check that a raw option really got accepted. This code places the value in the variable TcpLinger2:

```
      {ok,[{raw,6,8,<<TcpLinger2:32/native>>}]}=inet:getopts(Sock,[{raw,6,8,4}]),
```

Code such as the examples above is inherently non portable, even different versions of the same OS on the same platform may respond differently to this kind of option manipulation. Use with care.

Note that the default options for TCP/IP sockets can be changed with the Kernel configuration parameters mentioned in the beginning of this document.

POSIX Error Codes

- `e2big` - argument list too long
- `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` - RPC structure is bad
- `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 w/o 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` - file name too long
- `enavail` - not available
- `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 be 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` - program version wrong

- `eprogunavail` - RPC program not available
- `eproto` - protocol error
- `eprotonosupport` - protocol not supported
- `eprototype` - protocol wrong type for socket
- `erange` - math result unrepresentable
- `erefused` - EREFUSED
- `eremchg` - remote address changed
- `eremdev` - remote device
- `eremote` - pathname hit remote file system
- `eremoteio` - remote i/o error
- `eremoterelease` - EREMOTERELEASE
- `erofs` - read-only file system
- `erpcmismatch` - RPC version is wrong
- `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` - the hostname or domain name could not be found

inet_res

Erlang module

Performs DNS name resolving towards recursive name servers

See also *ERTS User's Guide: Inet configuration* for more information on how to configure an Erlang runtime system for IP communication and how to enable this DNS client by defining `'dns'` as a lookup method. It 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. It is just a DNS client that relies on asking trusted recursive nameservers.

Name Resolving

UDP queries are used unless resolver option `usevc` is `true`, which forces TCP queries. If the query is to 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 i.e `0` instead of `false`), resolver option `udp_payload_size` sets the limit. If a nameserver replies with the TC bit set (truncation), indicating the answer is incomplete, the query is retried to that nameserver using TCP. The resolver option `udp_payload_size` also sets the advertised size for the max allowed reply size, if EDNS is enabled, otherwise the nameserver uses the limit 512 byte. If the reply is larger it gets truncated, forcing a TCP re-query.

For UDP queries, the resolver options `timeout` and `retry` control retransmission. Each nameserver in the `nameservers` list is tried with a timeout of `timeout` / `retry`. Then all nameservers are tried again doubling the timeout, for a total of `retry` times.

For queries that not use the `search` list, if the query to all `nameservers` results in `{error,nxdomain}`or an empty answer, the same query is tried for the `alt_nameservers`.

Data Types

Resolver 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:

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 that return lists of {Field,Value} tuples. The arity 2 functions just return the value for a given 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 info 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) will convert
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()`

Resolve a DNS record of the given type for the given host, of class `in`. On success returns a `hostent()` record with `dns_data()` elements in the address list field.

This function uses the 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 simply 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:family_option()
Reason = inet:posix() | res_error()
```

Backend functions used by *inet:gethostbyname/1,2* .

This function uses the resolver option `search` just like *getbyname/2,3*.

If the resolver option `inet6` is `true`, an IPv6 address is looked up, and 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()
```

Resolve the DNS data for the record of the given type and class for the given 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` will give an empty answer since the answer records have specific types that are not `any`. An empty answer as well as a failed lookup returns an empty list.

Calls *resolve/2..4* with the same arguments and filters the result, so `Opts` is explained there.

```
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()*

Resolve a DNS record of the given type and class for the given name. The returned `dns_msg()` can be examined using access functions in `inet_db` as described 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` override the corresponding resolver options. If the option `nameservers` is given, it is also assumed that it is the complete list of nameserves, so the resolver option `alt_nameserves` is ignored. Of course, if that option is also given to this function, it is used.

The `verbose` option (or rather `{verbose,true}`), causes diagnostics printout through *io:format/2* of queries, replies retransmissions, etc, similar to from utilities like `dig`, `nslookup` et.al.

If `Opt` is an arbitrary 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` an alias for `{usevc,false}`.

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

Examples

Access functions example: how *lookup/3* could have been 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 have been deprecated due to the annoying double meaning of the nameservers/timeout argument, and because they had 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()
```

Resolve a DNS record of the given type and class for the given 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()
```

Resolve a DNS record of the given type and class for the given name.

init

Erlang module

The module init is moved to the runtime system application. Please see *init(3)* in the erts reference manual instead.

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 the `.hosts.erlang` file, see the section *Files* below. Returns the hosts in this file as a list, or returns `{error, Reason}` if the file could not be read or the Erlang terms on the file could not be interpreted.

localhost() -> Name

Types:

Name = string()

Returns the name of the local host. If Erlang was started with the `-name` command line flag, `Name` is the fully qualified name.

names() -> {ok, [{Name, Port}]} | {error, Reason}

names(Host) -> {ok, [{Name, Port}]} | {error, Reason}

Types:

Host = atom() | string()
Name = string()
Port = integer() >= 0
Reason = address | *file:posix()*

Similar to `epmd -names`, see `epmd(1)`. `Host` defaults to the local host. Returns the names and associated port numbers of the Erlang nodes that `epmd` at the specified host has registered.

Returns `{error, address}` if `epmd` is not running.

```
(arne@dunn)1> net_adm:names().
```

```
{ok,[{"arne",40262}]}
```

ping(Node) -> pong | pang

Types:

Node = atom()

Tries to set up a connection to `Node`. Returns `pang` if it fails, or `pong` if it is successful.

world() -> [node()]

world(Arg) -> [node()]

Types:

Arg = *verbosity()*

verbosity() = silent | verbose

This function calls `names(Host)` for all hosts which 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 were, 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 nodes in the network are not initially known.

Failure: `{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

As `world/0,1`, but the hosts are given as argument instead of being read from `.hosts.erlang`.

Files

The `.hosts.erlang` file 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 the `.hosts.erlang` file must be one host name per line. The host names must be within quotes as shown in the following 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 running 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 the 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 the command line flag `-sname`, the node name will be `foobar@Host`, where `Host` is the short name of the host (not the fully qualified domain name). If started with the `-name` flag, `Host` is the fully qualified domain name. See `erl(1)`.

Normally, connections are established automatically when another node is referenced. This functionality can be disabled by setting the Kernel configuration parameter `dist_auto_connect` to `false`, see *kernel(6)*. In this case, connections must be established explicitly by calling `net_kernel:connect_node/1`.

Which nodes are allowed to communicate with each other is handled by the magic cookie system, see *Distributed Erlang* in the Erlang Reference Manual.

Exports

`allow(Nodes) -> ok | error`

Types:

`Nodes = [node()]`

Limits access to the specified set of nodes. Any access attempts made from (or to) nodes not in `Nodes` will be rejected.

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.

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 process 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 of `kernel` version 2.11.4, and `erts` version 5.5.4, the following is guaranteed:

- `nodeup` messages will be delivered before delivery of any message from the remote node passed through the newly established connection.
- `nodedown` messages will not be delivered until all messages from the remote node that have been passed through the connection have been delivered.

Note, that this is *not* guaranteed for `kernel` versions before 2.11.4.

As of `kernel` version 2.11.4 subscriptions can also be made before the `net_kernel` server has been started, i.e., `net_kernel:monitor_nodes/[1,2]` does not return `ignored`.

As of `kernel` version 2.13, and `erts` version 5.7, the following is guaranteed:

- `nodeup` messages will be delivered after the corresponding node appears in results from `erlang:nodes/X`.
- `nodedown` messages will be delivered after the corresponding node has disappeared in results from `erlang:nodes/X`.

Note, 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:

```
{nodeup, Node} | {nodedown, Node}
  Node = node()
```

If `Options /= []`, the format is:

```
{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 *nodes/0*, are monitored.

`Option` can be any of the following:

`{node_type, NodeType}`

Currently valid values for `NodeType` are:

`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:

`connection_setup_failed`
The connection setup failed (after `nodeup` messages had been sent).

`no_network`
No network 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 timeout.

`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.

`get_net_ticktime() -> Res`

Types:

```
Res = NetTicktime | {ongoing_change_to, NetTicktime} | ignored
NetTicktime = integer() >= 1
```

Gets `net_ticktime` (see *kernel(6)*).

Currently defined return values (`Res`):

`NetTicktime`

`net_ticktime` is `NetTicktime` seconds.

`{ongoing_change_to, NetTicktime}`

`net_kernel` is currently changing `net_ticktime` to `NetTicktime` seconds.

```
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:

The minimum transition traffic interval (`MTTI`)

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

The transition period

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

If `<anno>NetTicktime</anno> < PreviousNetTicktime`, the actual `net_ticktime` change will be done at the end of the transition period; otherwise, at the beginning. During the transition period, `net_kernel` will ensure that there will be outgoing traffic on all connections at least every `MTTI` millisecond.

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

Returns one of the following:

`unchanged`

`net_ticktime` already had the value of `NetTicktime` and was left unchanged.

`change_initiated`

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

`{ongoing_change_to, NewNetTicktime}`

The request was *ignored*; because, `net_kernel` was busy changing `net_ticktime` to `NewNetTicktime` seconds.

```
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()

Note that the argument is a list with exactly one, two or three arguments. `NameType` defaults to `longnames` and `Ticktime` to 15000.

Turns a non-distributed node into a distributed node by starting `net_kernel` and other necessary processes.

`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 returns `{error, not_allowed}`. 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 will result 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.

Exports

cmd(Command) -> string()

Types:

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

Executes `Command` in a command shell of the target OS, captures the standard output of the command and returns this result as a string. This function is a replacement of the previous `unix:cmd/1`; on a Unix platform they are equivalent.

Examples:

```
LsOut = os:cmd("ls"), % on unix platform
DirOut = os:cmd("dir"), % on Win32 platform
```

Note that in some cases, standard output of a command when called from another program (for example, `os:cmd/1`) may differ, compared to the standard output of the command when called directly from an OS command shell.

find_executable(Name) -> Filename | false

find_executable(Name, Path) -> Filename | false

Types:

Name = Path = Filename = string()

These two functions look up an executable program given its name and a search path, in the same way as the underlying operating system. `find_executable/1` uses the current execution path (that is, the environment variable PATH on Unix and Windows).

`Path`, if given, should conform to the syntax of execution paths on the operating system. The absolute filename of the executable program `Name` is returned, or `false` if the program was not found.

getenv() -> [string()]

Returns a list of all environment variables. Each environment variable is given as a single string on the format `"VarName=Value"`, where `VarName` is the name of the variable and `Value` its value.

getenv(VarName) -> Value | false

Types:

VarName = string()

Value = string()

Returns the `Value` of the environment variable `VarName`, or `false` if the environment variable is undefined.

getpid() -> Value

Types:

Value = string()

Returns the process identifier of the current Erlang emulator in the format most commonly used by the operating system environment. `Value` is returned 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 VxWorks, `Value` contains the task id (decimal notation) of the Erlang task. On Windows, the process id as returned by the `GetCurrentProcessId()` system call is used.

putenv(VarName, Value) -> true

Types:

VarName = string()

Value = string()

Sets a new `Value` for the environment variable `VarName`.

timestamp() -> {MegaSecs, Secs, MicroSecs}

Types:

MegaSecs = Secs = MicroSecs = integer() >= 0

Returns a tuple in the same format as *erlang:now/0*. The difference is that this function returns what the operating system thinks (a.k.a. the wall clock time) without any attempts at time correction. The result of two different calls to this function is *not* guaranteed to be different.

The most obvious use for this function is logging. The tuple can be used together with the 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 timestamps in high resolution and consistent with the time in the rest of the operating system.

Example of code formatting a string in the 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]).
```

The module above could be used in the following way:

```
1> io:format("~s~n",[print_time:format_utc_timestamp()]).
29 Apr 2009  9:55:30.051711
```

type() -> vxworks | {Osfamily, Osname}

Types:

```
Osfamily = unix | win32
Osname = atom()
```

Returns the `Osfamily` and, in some cases, `Osname` of the current operating system.

On Unix, `Osname` will have same value as `uname -s` returns, but in lower case. For example, on Solaris 1 and 2, it will be `sunos`.

In Windows, `Osname` will be either `nt` (on Windows NT), or `windows` (on Windows 95).

On VxWorks the OS family alone is returned, that is `vxworks`.

> Note:
>
> Think twice before using this function. Use the `filename` module if you want to inspect or build file names in a portable way. Avoid matching on the `Osname` atom.

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

Types:

```
VersionString = string()
Major = Minor = Release = integer() >= 0
```

Returns the operating system version. On most systems, this function returns a tuple, but a string will be returned instead if the system has versions which 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. The groups in this module differ from the groups in the module `pg` in several ways. In `pg`, each message is sent to all members in the group. In this module, each message may be sent to one, some, or all 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) which are all members of the group `foobar`. There are no special functions for sending a message to the group. Instead, client functions should be written with the functions `get_members/1` and `get_local_members/1` to find out which processes are members of the group. Then the message can be sent to one or more members of the group.

If a member terminates, it is automatically removed from the group.

> Warning:
>
> This module is used by the `disk_log` module for managing distributed disk logs. The disk log names are used as group names, which means that some action may need to be taken 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}`

This is a useful dispatch function which can be used from client functions. It returns a process on the local node, if such a process exist. Otherwise, it chooses one randomly.

`get_members(Name) -> [pid()] | {error, {no_such_group, Name}}`

Types:

Name = *name()*

Returns all processes in the group `Name`. This function should be used from within a client function that accesses the group. It is therefore optimized for speed.

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 should 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 several times; it 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.

which_groups() -> [Name :: *name()*]

Returns a list of all known groups.

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 should be started explicitly. Use configuration parameters for `kernel` for this.

See Also

kernel(6), pg(3)

rpc

Erlang module

This module contains services which are 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

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 the 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 the node `Node` and returns the corresponding value `Res`, or `{badrpc, Reason}` if the call fails. `Timeout` is a timeout value in milliseconds. If the call times out, `Reason` is `timeout`.

If the reply arrives after the call times out, no message will contaminate the caller's message queue, since 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.

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

Types:

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

Like `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()
```

Like `block_call/4`, but with a timeout value in the same manner as `call/5`.

```
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 which does not suspend the caller until the result is finished. Instead, a key is returned which can be used at a later stage 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 the node `Node`.

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

Types:

```
Key = key()
Val = (Res :: term()) | {badrpc, 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`.

```
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()}

This is a non-blocking version of `yield/1`. It returns the tuple `{value, Val}` when the computation has finished, or `timeout` when `Timeout` milliseconds has elapsed.

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 which is sent concurrently from one client to multiple servers. This is useful for collecting some 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 also indicates some side effects RPCs may 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, Bin, File,]),

%% and then maybe check the ResL list.
```

cast(Node, Module, Function, Args) -> true

Types:

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

Evaluates `apply(Module, Function, Args)` on the 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.

`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.

`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 which 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 actually processed the message.

Any further messages sent to the servers, after this function has returned, will be 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

This function can be used when interacting with a server called `Name` at 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}`.

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()]

This function 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 a list of the nodes which did not exist, or where the server did not exist, or where the server terminated before sending any reply.

safe_multi_server_call(Name, Msg) -> {Replies, BadNodes}

safe_multi_server_call(Nodes, Name, Msg) -> {Replies, BadNodes}

Types:

```
Nodes = [node()]
Name = atom()
Msg = term()
Replies = [Reply :: term()]
BadNodes = [node()]
```

> Warning:
>
> This function is deprecated. Use `multi_server_call/2,3` instead.

In Erlang/OTP R6B and earlier releases, `multi_server_call/2,3` could not handle the case where the remote node exists, but there is no server called `Name`. Instead this function had to be used. In Erlang/OTP R7B and later releases, however, the functions are equivalent, except for this function being slightly slower.

`parallel_eval(FuncCalls) -> ResL`

Types:

```
FuncCalls = [{Module, Function, Args}]
Module = module()
Function = atom()
Args = ResL = [term()]
```

For every tuple in `FuncCalls`, evaluates `apply(Module, Function, Args)` on some node in the network. Returns the list of return values, in the same order as in `FuncCalls`.

`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`.

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

Types:

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

Location transparent version of the BIF *`process_info/1`*.

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

Types:

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

Location transparent version of the BIF *process_info/2*.

seq_trace

Erlang module

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

`seq_trace` provides functions which control all aspects of sequential tracing. There are functions for activation, deactivation, inspection and for collection of the trace output.

> ### Note:
>
> The implementation of sequential tracing is in beta status. This means that the programming interface still might undergo minor adjustments (possibly incompatible) based on feedback from users.

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
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`.

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
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() | false
```

Sets the system tracer. The system tracer can be either a process or port 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() | false
```

Returns the pid or port identifier 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 are:

```
{seq_trace, Label, SeqTraceInfo, TimeStamp}
```

or

```
{seq_trace, Label, SeqTraceInfo}
```

depending on whether the `timestamp` flag of the trace token is set to `true` or `false`. Where:

```
Label = int()
TimeStamp = {Seconds, Milliseconds, Microseconds}
  Seconds = Milliseconds = Microseconds = int()
```

The `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 the `'receive'` flag 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 the `print` flag set to `true` and `label` set to `Label`.

`Serial` is a tuple `{PreviousSerial, ThisSerial}`, where the first integer `PreviousSerial` denotes the serial counter passed in the last received message which carried a trace token. If the process is the first one in a new sequential trace, `PreviousSerial` is set to the value of the process internal "trace clock". The second integer `ThisSerial` is the serial counter that a process sets on outgoing messages and 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.

What is 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 one single message. In short it works like this:

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.

In order 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 will only receive those trace events that occur locally within the Erlang node. To get the whole picture of a sequential trace that involves processes on several Erlang nodes, the output from the system tracer on each involved node must be merged (off line).

In the following sections Sequential Tracing and its most fundamental concepts are described.

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 will be sent "invisibly" along with the message.

The current token of a process is set in two ways, either

- explicitly by the process itself, through a call to `seq_trace:set_token`, or
- when a message is received.

In both cases the current token will be set. In particular, if the token of a message received 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 1 and 2 above.

Serial

The trace token contains a component which is called `serial`. It consists of two integers `Previous` and `Current`. The purpose is to uniquely identify each traced event within a trace sequence and 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` which 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 same algorithm as for send above.
- *When a message is received and contains a nonempty 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
  ```

The `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 it will not be possible to use the serial for ordering of the trace events. To prevent the counter from overflowing in the middle of a sequential trace the function `seq_trace:reset_trace/0` can be called to reset the `prev_cnt` and `curr_cnt` of all processes in the Erlang node. This function will also set all trace tokens in processes and their message queues to empty and will thus stop all ongoing sequential tracing.

Performance considerations

The performance degradation for a system which is enabled for Sequential Tracing is negligible as long as no tracing is activated. When tracing is activated there will of course be an extra cost for each traced message but all other messages will be 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 has to 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 will appear as one process from the sequential tracing point of view.

In order to be able to perform sequential tracing between distributed Erlang nodes, the distribution protocol has been extended (in a backward compatible way). An Erlang node which supports sequential tracing can communicate with an older (OTP R3B) node but messages passed within that node can of course not be traced.

Example of Usage

The example shown here will give rough idea of how the new primitives can be used and what kind of output it will produce.

Assume that we 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 (inspired by AXE-10 and MD-110) could look like:

```
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 the printout above could 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 the tracer function above and sets that process as the system tracer could 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` below 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 which responds to all the 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

`wrap_log_reader` is a function to read internally formatted wrap disk logs, refer to disk_log(3). `wrap_log_reader` does not interfere with disk_log activities; there is however a known bug in this version of the `wrap_log_reader`, see chapter `bugs` below.

A wrap disk log file consists of several files, called index files. A log file can be opened and closed. It is also possible to open just one index file separately. If an non-existent or a non-internally formatted file is opened, an error message is returned. If the file is corrupt, no attempt to repair it will be done but an error message is returned.

If a log is configured to be distributed, there is a possibility that all items are not loggen on all nodes. `wrap_log_reader` does only read the log on the called node, it is entirely 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()}
```

This function makes it possible to efficiently read the terms which have been appended to a log. It minimises disk I/O by reading large 8K chunks from the file.

The first time `chunk` is called an initial continuation returned from the `open/1`, `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. Default is `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.

The `chunk` function 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 into any subsequent calls to `chunk`. With a series of calls to `chunk` it is then possible to extract all terms from a log.

The `chunk` function 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. Note also that the log is not repaired.

`chunk` returns `{Continuation2, eof}` when the end of the log is reached, and `{error, Reason}` if an error occurs.

The returned continuation may or may not be valid in the next call to `chunk`. This is because the log may wrap and delete the file into which the continuation points. To make sure this does not happen, the log can be blocked during the search.

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

Types:

`Continuation = `*`continuation()`

`Reason = `*`file:posix()`

This function 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 which is to be read.

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

The `open` function returns `{ok, Continuation}` if the log/index file was successfully opened. The `Continuation` is to be used when chunking or closing the file.

The function returns `{error, Reason}` for all errors.

Bugs

This version of the `wrap_log_reader` does not detect if the `disk_log` wraps to a new index file between a `wrap_log_reader:open` and the first `wrap_log_reader:chunk`. In this case the chuck will actually read the last logged items in the log file, because the opened index file was truncated by the `disk_log`.

See Also

disk_log(3)

zlib

Erlang module

The module zlib is moved to the runtime system application. Please see *zlib(3)* in the erts reference manual instead.

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 should be called `Application.app` where `Application` is the name of the application. The file should be located in the `ebin` directory for the application.

It must contain one 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}]}.

             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}  undefined
Phases       [{Phase,PhaseArgs}] undefined
  Module = Name = App = Par = Phase = atom()
  Val = StartArgs = PhaseArgs = term()
```

`Application` is the name of the application.

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` and `applications`. The other keys are ignored by `systools`.

`description`

A one-line description of the application.

`id`

Product identification, or similar.

`vsn`

The 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 - will be ignored
The maximum number of processes allowed in the application.

`maxT`

The maximum time in milliseconds that the application is allowed to run. After the specified time the application will automatically terminate.

`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 which are included by this application. When this application is started, all included application will automatically be loaded, but not started, by the application controller. It is assumed that the topmost supervisor of the included application is started by a supervisor of this application.

`applications`

All applications which 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 note 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 `erl(1)`).

`mod`

Specifies the application callback module and a start argument, see `application(3)`.

The `mod` key is necessary for an application implemented as a supervision tree, or the application controller will not know how to start it. The `mod` key can be omitted for applications without processes, typically code libraries such as the application STDLIB.

`start_phases`

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

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

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

The application master will then call `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 application, 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. Refer to *OTP Design Principles* for more information.

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 will 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 `erl(1)`).

The value of a configuration parameter is retrieved by calling `application:get_env/1,2`.

FILE SYNTAX

The configuration file should be called `Name.config` where `Name` is an arbitrary name.

The `.config` file contains one single Erlang term. The file has the following syntax:

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

- `Application = atom()` is the name of the application.
- `Par = atom()` is the name of a configuration parameter.
- `Val = term()` is the 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 should 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, or additional, `.config` files would lead to inconsistent update of application configurations. Therefore, in Erlang 5.4/OTP R10B, the syntax of `sys.config` was extended to allow pointing out other `.config` files:

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

- `File = string()` is the name of another `.config` file. The extension `.config` may 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 will yield the following environment for `myapp`:

```
[{par1,val1},{par2,val3},{par3,val4}]
```

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

SEE ALSO

`app(4)`, `erl(1)`, *OTP Design Principles*

packages

Erlang module

> ## Warning:
>
> Packages has since it was introduced more than 5 years ago been an experimental feature. Use it at your own risk, we do not actively maintain and develop this feature. It might however be supported some day.
>
> In spite of this packages work quite well, but there are some known issues in tools and other parts where packages don't work well.

Introduction

Packages are simply namespaces for modules. All old Erlang modules automatically belong to the top level ("empty-string") namespace, and do not need any changes.

The full name of a packaged module is written as e.g. "`fee.fie.foe.foo`", i.e., as atoms separated by periods, where the package name is the part up to but not including the last period; in this case "`fee.fie.foe`". A more concrete example is the module `erl.lang.term`, which is in the package `erl.lang`. Package names can have any number of segments, as in `erl.lang.list.sort`. The atoms in the name can be quoted, as in `foo.'Bar'.baz`, or even the whole name, as in `'foo.bar.baz'` but the concatenation of atoms and periods must not contain two consecutive period characters or end with a period, as in `'foo..bar'`, `foo.'.bar'`, or `foo.'bar.'`. The periods must not be followed by whitespace.

The code loader maps module names onto the file system directory structure. E.g., the module `erl.lang.term` corresponds to a file `.../erl/lang/term.beam` in the search path. Note that the name of the actual object file corresponds to the last part only of the full module name. (Thus, old existing modules such as `lists` simply map to `.../lists.beam`, exactly as before.)

A packaged module in a file "`foo/bar/fred.erl`" is declared as:

```
-module(foo.bar.fred).
```

This can be compiled and loaded from the Erlang shell using `c(fred)`, if your current directory is the same as that of the file. The object file will be named `fred.beam`.

The Erlang search path works exactly as before, except that the package segments will be appended to each directory in the path in order to find the file. E.g., assume the path is `["/usr/lib/erl", "/usr/local/lib/otp/legacy/ebin", "/home/barney/erl"]`. Then, the code for a module named `foo.bar.fred` will be searched for first as `"/usr/lib/erl/foo/bar/fred.beam"`, then `"/usr/local/lib/otp/legacy/ebin/foo/bar/fred.beam"` and lastly `"/home/barney/erl/foo/bar/fred.beam"`. A module like `lists`, which is in the top-level package, will be looked for as `"/usr/lib/erl/lists.beam"`, `"/usr/local/lib/otp/legacy/ebin/lists.beam"` and `"/home/barney/erl/lists.beam"`.

Programming

Normally, if a call is made from one module to another, it is assumed that the called module belongs to the same package as the source module. The compiler automatically expands such calls. E.g., in:

```
-module(foo.bar.m1).
```

```
-export([f/1]).

f(X) -> m2:g(X).
```

`m2:g(X)` becomes a call to `foo.bar.m2` If this is not what was intended, the call can be written explicitly, as in

```
-module(foo.bar.m1).
-export([f/1]).

f(X) -> fee.fie.foe.m2:g(X).
```

Because the called module is given with an explicit package name, no expansion is done in this case.

If a module from another package is used repeatedly in a module, an import declaration can make life easier:

```
-module(foo.bar.m1).
-export([f/1, g/1]).
-import(fee.fie.foe.m2).

f(X) -> m2:g(X).
g(X) -> m2:h(X).
```

will make the calls to `m2` refer to `fee.fie.foe.m2`. More generally, a declaration `-import(Package.Module).` will cause calls to `Module` to be expanded to `Package.Module`.

Old-style function imports work as normal (but full module names must be used); e.g.:

```
-import(fee.fie.foe.m2, [g/1, h/1]).
```

however, it is probably better to avoid this form of import altogether in new code, since it makes it hard to see what calls are really "remote".

If it is necessary to call a module in the top-level package from within a named package, the module name can be written either with an initial period as in e.g. "`.lists`", or with an empty initial atom, as in "`''.lists`". However, the best way is to use an import declaration - this is most obvious to the eye, and makes sure we don't forget adding a period somewhere:

```
-module(foo.bar.fred).
-export([f/1]).
-import(lists).

f(X) -> lists:reverse(X).
```

The dot-syntax for module names can be used in any expression. All segments must be constant atoms, and the result must be a well-formed package/module name. E.g.:

```
spawn(foo.bar.fred, f, [X])
```

is equivalent to `spawn('foo.bar.fred', f, [X])`.

The Erlang Shell

The shell also automatically expands remote calls, however currently no expansions are made by default. The user can change the behaviour by using the `import/1` shell command (or its abbreviation `use/1`). E.g.:

```
1> import(foo.bar.m).
ok
2> m:f().
```

will evaluate `foo.bar.m:f()`. If a new import is made of the same name, this overrides any previous import. (It is likely that in the future, some system packages will be pre-imported.)

In addition, the shell command `import_all/1` (and its alias `use_all/1`) imports all modules currently found in the path for a given package name. E.g., assuming the files "`.../foo/bar/fred.beam`", "`.../foo/bar/barney.beam`" and "`.../foo/bar/bambam.beam`" can be found from our current path,

```
1> import_all(foo.bar).
```

will make `fred`, `barney` and `bambam` expand to `foo.bar.fred`, `foo.bar.barney` and `foo.bar.bambam`, respectively.

Note: The compiler does not have an "import all" directive, for the reason that Erlang has no compile time type checking. E.g. if the wrong search path is used at compile time, a call `m:f(...)` could be expanded to `foo.bar.m:f(...)` without any warning, instead of the intended `frob.ozz.m:f(...)`, if package `foo.bar` happens to be found first in the path. Explicitly declaring each use of a module makes for safe code.

Exports

`no functions exported`