Kernel Application (KERNEL)

version 2.10

Typeset in LaTeX from SGML source using the DOCBUILDER 3.3.2 Document System.

Contents

Kernel Reference Manual

Short Summaries

kernel

No functions are exported.

application

The following functions are exported:

- `get_all_env() -> Env`
  [page 33] Get the configuration parameters for an application.
- `get_all_env(Application) -> Env`
  [page 33] Get the configuration parameters for an application.
- `get_all_key() -> {ok, Keys} | []`
  [page 33] Get the application specification keys.
- `get_all_key(Application) -> {ok, Keys} | undefined`
  [page 33] Get the application specification keys.
- `get_application() -> {ok, Application} | undefined`
  [page 34] Get the name of an application containing a certain process or module.
- `get_application(Pid | Module) -> {ok, Application} | undefined`
  [page 34] Get the name of an application containing a certain process or module.
- `get_env(Par) -> {ok, Val} | undefined`
  [page 34] Get the value of a configuration parameter.
- `get_env(Application, Par) -> {ok, Val} | undefined`
  [page 34] Get the value of a configuration parameter.
- `get_key(Key) -> {ok, Val} | undefined`
  [page 34] Get the value of an application specification key.
- `get_key(Application, Key) -> {ok, Val} | undefined`
  [page 34] Get the value of an application specification key.
- `load(AppDescr) -> ok | {error, Reason}`
  [page 34] Load an application.
- `load(AppDescr, Distributed) -> ok | {error, Reason}`
  [page 34] Load an application.
- `loaded_applications() -> [{Application, Description, Vsn}]`
  [page 35] Get the currently loaded applications.
- `permit(Application, Bool) -> ok | {error, Reason}`
  [page 35] Change an application's permission to run on a node.
- `set_env(Application, Par, Val) -> ok`
  [page 36] Set the value of a configuration parameter.
- `start(Application) -> ok | {error, Reason}`
  [page 36] Load and start an application.
- `start(Application, Type) -> ok | {error, Reason}`
  [page 36] Load and start an application.
- `start_type() -> StartType | local | undefined`
  [page 37] Get the start type of an ongoing application startup.
- `stop(Application) -> ok | {error, Reason}`
  [page 37] Stop an application.

- takeover(Application, Type) -> ok | {error, Reason}
  [page 38] Take over a distributed application.
- unload(Application) -> ok | {error, Reason}
  [page 38] Unload an application.
- unset_env(Application, Par) -> ok
  [page 38] Unset the value of a configuration parameter.
- which_applications() -> [{Application, Description, Vsn}]
  [page 39] Get the currently running applications.
- Module:start(StartType, StartArgs) -> {ok, Pid} | {ok, Pid, State} | {error, Reason}
  [page 39] Start an application.
- Module:start_phase(Phase, StartType, PhaseArgs) -> ok | {error, Reason}
  [page 40] Extended start of an application.
- Module:prep_stop(State) -> NewState
  [page 40] Prepare an application for termination.
- Module:stop(State)
  [page 40] Clean up after termination of an application.
- Module:config_change(Changed, New, Removed) -> ok
  [page 41] Update the configuration parameters for an application.

auth

The following functions are exported:

- start()
  [page 43] Start the auth server
- stop()
  [page 43] Stop the auth server
- is_auth(Node)
  [page 43] Return status of communication authorization
- exists(Node)
  [page 43] Check if a node exists
- cookie()
  [page 43] Read and set cookies
- node_cookie(Node, Cookie)
  [page 43] Check if a cookie is known
- node_cookie([Node, Cookie])
  [page 43] Check if a cookie is known
- cookie([Cookie])
  [page 43] Set the distribution cookie for the local node

code

The following functions are exported:

- `start() -> {ok, Pid} | {error, What}`
  [page 44] Start the code server.
- `start(Flags) -> {ok, Pid} | {error, What}`
  [page 44] Start the code server.
- `start_link() -> {ok, Pid} | {error, What}`
  [page 45] Start and links to the code server.
- `start_link(Flags) -> {ok, Pid} | {error, What}`
  [page 45] Start and links to the code server.
- `set_path(DirList) -> true | {error, What}`
  [page 45] Set the code server search path.
- `get_path() -> Path`
  [page 45] Return the current path of the code server.
- `add_path(Dir) -> true | {error, What}`
  [page 45] Add a directory to the end of path.
- `add_pathz(Dir) -> true | {error, What}`
  [page 45] Add a directory to the end of path.
- `add_patha(Dir) -> true | {error, What}`
  [page 45] Add a directory to the beginning of path.
- `add_paths(DirList) -> ok`
  [page 46] Add directories to the end of path.
- `add_pathsz(DirList) -> ok`
  [page 46] Add directories to the end of path.
- `add_pathsa(DirList) -> ok`
  [page 46] Add directories to the beginning of path.
- `del_path(NameDir) -> true | false | {error, What}`
  [page 46] Delete a directory from the path.
- `replace_path(Name, Dir) -> true | {error, What}`
  [page 46] Replace a directory with another in the path.
- `load_file(Module) -> {module, Module} | {error, What}`
  [page 46] Load a module (residing in `File`).
- `load_abs(File) -> {module, Module} | {error, What}`
  [page 47] Load a module (residing in `File`).
- `ensure_loaded(Module) -> {module, Module} | {error, What}`
  [page 47] Try to ensure that a module is loaded.
- `delete(Module) -> true | false`
  [page 47] Delete the code in `Module`.
- `purge(Module) -> true | false`
  [page 47] Purges the code in `Module`.
- `soft_purge(Module) -> true | false`
  [page 48] Purge the code in `Module` if no process uses it.
- `is_loaded(Module) -> {file, Loaded} | false`
  [page 48] Test if `Module` is loaded.

- `all_loaded() -> [LoadMod]`
  [page 48] Get all loaded modules.
- `load_binary(Module, File, Binary) -> {module, Module} | {error, What}`
  [page 48] Load object code as a binary.
- `stop() -> stopped`
  [page 48] Stop the code server.
- `root_dir() -> RootDir`
  [page 48] Return the root directory of Erlang/OTP.
- `lib_dir() -> LibDir`
  [page 49] Return the library directory.
- `lib_dir(Name) -> LibDir | {error, What}`
  [page 49] Return the directory for name.
- `compiler_dir() -> CompDir`
  [page 49] Return the compiler directory.
- `priv_dir(Name) -> PrivDir | {error, What}`
  [page 49] Return the priv directory for name.
- `get_object_code(Module) -> {Module, Bin, AbsFileName} | error`
  [page 49] Get the object code for a module.
- `objfile_extension() -> Ext`
  [page 50] Return the object code file extension.
- `rehash() -> ok`
  [page 50] Rehash or create code path cache.
- `stick_dir(Dir) -> ok | {error, term()}`
  [page 50] Mark a directory as 'sticky'.
- `unstick_dir(Dir) -> ok | {error, term()}`
  [page 50] Mark a directory as 'non-sticky'.
- `which(Module) -> WhichFile`
  [page 50] Return the directory of a module.
- `where_is_file(File) -> FullName`
  [page 50] Return the full name of any file located in a code path directory.
- `clash() -> ok`
  [page 50] Searche for modules with identical names.

disk_log

The following functions are exported:

- `accessible_logs() -> {[LocalLog], [DistributedLog]}`
  [page 54] Return the accessible disk logs on the current node.
- `alog(Log, Term)`
  [page 54] Asynchronously log an item onto a disk log.
- `balog(Log, Bytes) -> ok | {error, Reason}`
  [page 54] Asynchronously log an item onto a disk log.
- `alog_terms(Log, TermList)`
  [page 54] Asynchronously log several items onto a disk log.

- balog_terms(Log, BytesList) -> ok | {error, Reason}
  [page 54] Asynchronously log several items onto a disk log.
- block(Log)
  [page 55] Block a disk log.
- block(Log, QueueLogRecords) -> ok | {error, Reason}
  [page 55] Block a disk log.
- change_header(Log, Header) -> ok | {error, Reason}
  [page 55] Change the head or head_func option for an owner of a disk log.
- change_notify(Log, Owner, Notify) -> ok | {error, Reason}
  [page 55] Change the notify option for an owner of a disk log.
- change_size(Log, Size) -> ok | {error, Reason}
  [page 55] Change the size of an open disk log.
- chunk(Log, Continuation)
  [page 56] Read a chunk of objects written to a disk log.
- chunk(Log, Continuation, N) -> {Continuation2, Terms} |
  {Continuation2, Terms, Badbytes} | eof | {error, Reason}
  [page 56] Read a chunk of objects written to a disk log.
- bchunk(Log, Continuation)
  [page 56] Read a chunk of objects written to a disk log.
- bchunk(Log, Continuation, N) -> {Continuation2, Binaries} |
  {Continuation2, Binaries, Badbytes} | eof | {error, Reason}
  [page 56] Read a chunk of objects written to a disk log.
- chunk_info(Continuation) -> InfoList | {error, Reason}
  [page 57] Return information about a chunk continuation of a disk log.
- chunk_step(Log, Continuation, Step) -> {ok, Continuation2} | {error,
  Reason}
  [page 57] Step forward or backward among the wrap log files of a disk log.
- close(Log) -> ok | {error, Reason}
  [page 58] Close a disk log.
- format_error(Error) -> character_list()
  [page 58] Return an English description of a disk log error reply.
- inc_wrap_file(Log) -> ok | {error, Reason}
  [page 58] Change to the next wrap log file of a disk log.
- info(Log) -> InfoList | {error, no_such_log}
  [page 58] Return information about a disk log.
- lclose(Log)
  [page 59] Close a disk log on one node.
- lclose(Log, Node) -> ok | {error, Reason}
  [page 59] Close a disk log on one node.
- log(Log, Term)
  [page 60] Log an item onto a disk log.
- blog(Log, Bytes) -> ok | {error, Reason}
  [page 60] Log an item onto a disk log.
- log_terms(Log, TermList)
  [page 60] Log several items onto a disk log.
- blog_terms(Log, BytesList) -> ok | {error, Reason}
  [page 60] Log several items onto a disk log.

- `open(ArgL) -> OpenRet | DistOpenRet`
  [page 61] Open a disk log file.
- `pid2name(Pid) -> {ok, Log} | undefined`
  [page 64] Return the name of the disk log handled by a pid.
- `reopen(Log, File)`
  [page 64] Reopen a disk log and save the old log.
- `reopen(Log, File, Head)`
  [page 64] Reopen a disk log and save the old log.
- `breopen(Log, File, BHead) -> ok | {error, Reason}`
  [page 64] Reopen a disk log and save the old log.
- `sync(Log) -> ok | {error, Reason}`
  [page 64] Flush the contents of a disk log to the disk.
- `truncate(Log)`
  [page 65] Truncate a disk log.
- `truncate(Log, Head)`
  [page 65] Truncate a disk log.
- `btruncate(Log, BHead) -> ok | {error, Reason}`
  [page 65] Truncate a disk log.
- `unblock(Log) -> ok | {error, Reason}`
  [page 65] Unblock a disk log.

erl_boot_server

The following functions are exported:

- `start(Slaves) -> {ok, Pid} | {error, What}`
  [page 66] Start the boot server.
- `start_link(Slaves) -> {ok, Pid} | {error, What}`
  [page 66] Start the boot server and links the caller.
- `add_slave(Slave) -> ok | {error, What}`
  [page 66] Add a slave to the list of allowed slaves.
- `delete_slave(Slave) -> ok | {error, What}`
  [page 66] Delete a slave from the list of allowed slaves.
- `which_slaves() -> Slaves`
  [page 67] Return the current list of allowed slave hosts.

erl_ddll

The following functions are exported:

- `start() -> {ok, Pid} | {error, Reason}`
  [page 68] Start the server.
- `start_link() -> {ok, Pid} | {error, Reason}`
  [page 68] Start the server and links it to the calling process.
- `stop() -> ok`
  [page 68] Stop the server.
- `load_driver(Path, Name) -> ok | {error, ErrorDescriptor}`
  [page 68] Load a driver.

- `unload_driver(Name) -> ok | {error, ErrorDescriptor}`
  [page 68] Load a driver.
- `loaded_drivers() -> {ok, DriverList}`
  [page 69] List loaded drivers.
- `format_error(ErrorDescriptor) -> string()`
  [page 69] Format an error descriptor

erl_prim_loader

The following functions are exported:

- `start(Id,Loader,Hosts) -> {ok, Pid} | {error, What}`
  [page 71] Start the Erlang low level loader.
- `get_file(File) -> {ok, Bin, FullName} | error`
  [page 71] Get a file.
- `get_path() -> {ok, Path}`
  [page 72] Get the path set in the loader.
- `set_path(Path) -> ok`
  [page 72] Set the path of the loader.

erlang

The following functions are exported:

- `abs(Number)`
  [page 74] Arithmetical absolute value
- `erlang:append_element(Tuple, Term)`
  [page 74] Append an extra element to a tuple
- `apply(Fun, ArgumentList)`
  [page 74] Apply a function to an argument list
- `apply(Module, Function, ArgumentList)`
  [page 75] Apply a function to an argument list
- `atom_to_list(Atom)`
  [page 75] Convert an atom to a list
- `binary_to_list(Binary)`
  [page 75] Return a list of integers which correspond to the bytes of `Binary`
- `binary_to_list(Binary, Start, Stop)`
  [page 75] Return a list of integers which correspond to the bytes of `Binary`
- `binary_to_term(Binary)`
  [page 75] Return an Erlang term which is the result of decoding the binary `Binary`
- `erlang:bump_reductions(Reductions)`
  [page 76] Increment the reduction counter for the current process
- `erlang:cancel_timer(Ref)`
  [page 76] Cancel a timer
- `check_process_code(Pid, Module)`
  [page 76] Check if a process is executing old code
- `concat_binary(ListOfBinaries)`
  [page 77] Concatenate a list of binaries

- erlang:send(Dest, Msg, Options)
  [page 107] Send a message conditionally
- erlang:send_after(Time, Pid, Msg)
  [page 107] Send a message after a certain time
- erlang:send_nosuspend(Dest, Msg)
  [page 108] Try to send a message without ever blocking.
- erlang:send_nosuspend(Dest, Msg, Options)
  [page 108] Try to send a message without ever blocking.
- erlang:set_cookie(Node, Cookie)
  [page 109] Set the magic cookie of a node
- setelement(Index, Tuple, Value)
  [page 109] Set Nth element of a tuple
- size(Item)
  [page 109] Size of a tuple or binary
- spawn(Fun)
  [page 109] Create a new Erlang process with a fun as entry point
- spawn(Node, Fun)
  [page 109] Create a new Erlang process with a fun as entry point on a given node
- spawn(Module, Function, ArgumentList)
  [page 110] Create a new Erlang process with a specified function as entry point
- spawn(Node, Module, Function, ArgumentList)
  [page 110] Create a new Erlang process with a specified function as entry point on a given node
- spawn_link(Fun)
  [page 110] Create a new Erlang process with a fun as entry point and link to the new process
- spawn_link(Node, Fun)
  [page 110] Create a new Erlang process with a fun as entry point on a specified node and link to the new process
- spawn_link(Module, Function, ArgumentList)
  [page 110] Create a new Erlang process with a specified function as entry point and link to the new process
- spawn_link(Node, Module, Function, ArgumentList)
  [page 110] Create a new Erlang process on a specified node with a function as the entry point and link to the new process
- spawn_opt(Fun, Options)
  [page 111] Create a new Erlang process with a fun as entry point giving additional options
- spawn_opt(Node, Fun, Options)
  [page 111] Create a new Erlang process with a fun as entry point on a specified node giving additional options
- spawn_opt(Module, Function, ArgumentList, Options)
  [page 111] Create a new Erlang process with a function as entry point giving additional options
- spawn_opt(Node, Module, Function, ArgumentList, Options)
  [page 112] Create a new Erlang process on a specified node with a function as entry point giving additional options

- `erlang:universaltime_to_localtime(DateTime)`
  [page 125] Convert UTC date and time in to local date and time
- `unlink(Pid)`
  [page 125] Remove a link, if there is one, from the calling process to another process
- `unregister(Name)`
  [page 125] Remove the registered name for a process or port
- `whereis(Name)`
  [page 126] Return the pid or port corresponding to a registered name
- `erlang:yield()`
  [page 126] Let other processes get a chance to execute

error_handler

The following functions are exported:

- `undefined_function(Module, Func, ArgList) -> term()`
  [page 127] Called when an undefined function is encountered
- `undefined_lambda(Module, Fun, ArgList) -> term()`
  [page 127] Called when an undefined lambda (fun) is encountered

error_logger

The following functions are exported:

- `start() -> {ok, Pid} | {error, What}`
  [page 129] Start the error logger event manager.
- `start_link() -> {ok, Pid} | {error, What}`
  [page 129] Start the error logger event manager.
- `error_report(Report) -> ok`
  [page 129] Send a standard error report event to the error logger.
- `error_report(Type,Report) -> ok`
  [page 130] Send a user defined error report type event.
- `info_report(Report) -> ok`
  [page 130] Send an information report to the error logger.
- `info_report(Type,Report) -> ok`
  [page 130] Send a user defined information report type event.
- `error_msg(Format) -> ok`
  [page 131] Send an error event to the error logger.
- `error_msg(Format,Args) -> ok`
  [page 131] Send an error event to the error logger.
- `format(Format,Args) -> ok`
  [page 131] Send an error event to the error logger.
- `info_msg(Format) -> ok`
  [page 131] Send an information event to the error logger.
- `info_msg(Format,Args) -> ok`
  [page 131] Send an information event to the error logger.

- `tty(Flag) -> ok`
  [page 131] Enable or disables error printouts to the tty.
- `logfile(Request) -> ok | FileName | {error, What}`
  [page 131] Enable or disables error printouts to a file.
- `add_report_handler(Module) -> ok | Other`
  [page 132] Add a new event handler to the error logger.
- `add_report_handler(Module,Args) -> ok | Other`
  [page 132] Add a new event handler to the error logger.
- `delete_report_handler(Module) -> Return | {error, What}`
  [page 132] Delete an error report handler.
- `swap_handler(ToHandler) -> ok`
  [page 132] Swap from a primitive first handler to a standard event handler
- `warning_map() -> TagAtom`
  [page 132] Determines the current mapping for warning events
- `warning_msg(Format) -> ok`
  [page 132] Send an warning event to the error logger.
- `warning_msg(Format,Args) -> ok`
  [page 132] Send an warning event to the error logger.
- `warning_report(Report) -> ok`
  [page 133] Send a warning report type event.
- `warning_report(Type,Report) -> ok`
  [page 133] Send a warning report type event.

file

The following functions are exported:

- `change_group(Filename, Gid)`
  [page 135] Change owner for a file
- `change_owner(Filename, Uid)`
  [page 135] Change owner of a file
- `change_owner(Filename, Uid, Gid)`
  [page 135] Change owner for a file
- `change_time(Filename, Mtime)`
  [page 135] Change the modification time for a file
- `change_time(Filename, Mtime, Atime)`
  [page 135] Change the modification time for a file
- `close(IoDevice)`
  [page 135] Close a file
- `consult(Filename)`
  [page 136] Read Erlang terms from a file
- `copy(Source, Destination)`
  [page 136] Copies file contents
- `copy(Source, Destination, ByteCount)`
  [page 136] Copies file contents
- `del_dir(DirName)`
  [page 136] Delete a directory

- pwrite(IoDevice, Location, Bytes)
  [page 144] Write to a file at a certain position
- read(IoDevice, Number)
  [page 144] Read from a file
- read_file(Filename)
  [page 144] Read a file
- read_file_info(Filename)
  [page 144] Get information about a file
- read_link(Linkname)
  [page 145] See what a link is pointing to
- read_link_info(Filename)
  [page 146] Get information about a link or file
- rename(Source, Destination)
  [page 146] Rename a file
- script(Filename)
  [page 146] Evaluate and return the value of expressions in a file
- script(Filename, Bindings)
  [page 147] Evaluate and return the value of expressions in a file
- set_cwd(DirName)
  [page 147] Set the current working directory
- sync(IoDevice)
  [page 147] Synchronizes the in-memory state of a file with that on the physical medium
- truncate(IoDevice)
  [page 147] Truncate a file
- write(IoDevice, Bytes)
  [page 147] Write to a file
- write_file(Filename, Binary)
  [page 147] Write a file
- write_file(Filename, Binary, ModeList)
  [page 148] Write a file
- write_file_info(Filename, FileInfo)
  [page 148] Change file information

gen_tcp

The following functions are exported:

- accept(ListenSocket) -> {ok, Socket} | {error, Reason}
  [page 154] Accept an incoming connection request on a listen socket.
- accept(ListenSocket, Timeout) -> {ok, Socket} | {error, Reason}
  [page 154] Accept an incoming connection request on a listen socket.
- close(Socket) -> ok | {error, Reason}
  [page 154] Close an TCP socket
- connect(Address, Port, Options) -> {ok, Socket} | {error, Reason}
  [page 154] Connect to a TCP port.

- connect(Address, Port, Options, Timeout) -> {ok, Socket} | {error, Reason}
  [page 154] Connect to a TCP port.
- controlling_process(Socket, NewOwner) -> ok | {error, eperm}
  [page 155] Assign a new controlling process to a socket
- listen(Port, Options) -> {ok, Socket} | {error, Reason}
  [page 155] Set up a socket which listen on Port
- recv(Socket, Length) -> {ok, Packet} | {error, Reason}
  [page 156] Receive a packet from a passive socket
- recv(Socket, Length, Timeout)
  [page 156] Receive a packet from a passive socket
- send(Socket, Packet) -> ok | {error, Reason}
  [page 156] Send a packet
- shutdown(Socket, How) -> ok | {error, Reason}
  [page 157] Immediately close a socket in one direction

gen_udp

The following functions are exported:

- close(Socket) -> ok | {error, Reason}
  [page 158] Close Socket.
- controlling_process(Socket,NewOwner) ->
  [page 158] Change controlling process of a Socket.
- open(Port) -> {ok, Socket } | { error, Reason }
  [page 158] Associate a UDP port number with the process calling it.
- open(Port,Options) -> {ok, Socket } | { error, Reason }
  [page 158] Associate a UDP port number with the process calling it.
- recv(Socket, Length) -> {ok,{Address, Port, Packet}} | {error, Reason}
  [page 159] Receive a packet from a passive socket
- recv(Socket, Length, Timeout)
  [page 159] Receive a packet from a passive socket
- send(S,Address,Port,Packet) -> ok | {error, Reason}
  [page 159] Send a packet to a specified Address and Port (from port associated with Id).

global

The following functions are exported:

- del_lock(Id)
  [page 161] Delete the lock Id
- del_lock(Id, Nodes) -> void()
  [page 161] Delete the lock Id
- notify_all_name(Name, Pid1, Pid2) -> none
  [page 161] Name resolving function that notifies both Pids

- `random_exit_name(Name, Pid1, Pid2) -> Pid1 | Pid2`
  [page 161] Name resolving function that kills one Pid
- `random_notify_name(Name, Pid1, Pid2) -> Pid1 | Pid2`
  [page 161] Name resolving function that notifies one Pid
- `register_name(Name, Pid)`
  [page 161] Globally registers `Pid` as `Name`
- `register_name(Name, Pid, Resolve) -> yes | no`
  [page 161] Globally registers `Pid` as `Name`
- `registered_names() -> [Name]`
  [page 162] Return all globally registered names
- `re_register_name(Name, Pid)`
  [page 162] Atomically re-register `Pid` for `Name`
- `re_register_name(Name, Pid, Resolve) -> void()`
  [page 162] Atomically re-register `Pid` for `Name`
- `send(Name, Msg) -> Pid`
  [page 162] Send `Msg` to the global process `Name`
- `set_lock(Id)`
  [page 162] Set a lock on the specified nodes
- `set_lock(Id, Nodes)`
  [page 162] Set a lock on the specified nodes
- `set_lock(Id, Nodes, Retries) -> boolean()`
  [page 162] Set a lock on the specified nodes
- `start()`
  [page 163] Start the global name server
- `start_link() -> {ok, Pid} | {error, Reason}`
  [page 163] Start the global name server
- `stop() -> void()`
  [page 163] Stop the global name server
- `sync() -> void()`
  [page 163] Synchronize the global name server
- `trans(Id, Fun)`
  [page 163] Micro transaction facility
- `trans(Id, Fun, Nodes)`
  [page 163] Micro transaction facility
- `trans(Id, Fun, Nodes, Retries) -> Res | aborted`
  [page 163] Micro transaction facility
- `unregister_name(Name) -> void()`
  [page 164] Unregister the global name `Name`
- `whereis_name(Name) -> Pid() | undefined`
  [page 164] Return the Pid of the global process `Name`

global_group

The following functions are exported:

- `global_groups() -> {OwnGroupName, [OtherGroupName]} | undefined`
  [page 166] Return the global group names

- `info() -> [{state, State}, {own_group_name, atom()}, {own_group_nodes, [Node]}, {synced_nodes, [Node]}, {sync_error, [Node]}, {no_contact, [Node]}, {other_groups, Other_grps}, {monitoring, [pid()]}]`
  [page 166] Return the state of the global group process
- `monitor_nodes(Flag) -> ok`
  [page 167] Subscription of node status for nodes in the immediate global group
- `own_nodes() -> [Node] | {error, ErrorMsg}`
  [page 167] Return the global group names
- `registered_names({node, Node}) -> [Name] | {error, ErrorMsg}`
  [page 167] Return all globally registered names
- `registered_names({group, GlobalGroupName}) -> [Name]`
  [page 167] Return all globally registered names
- `send(Name, Msg) -> Pid | {badarg, Msg} | {error, ErrorMsg}`
  [page 167] Send `Msg` to a registered process `Name`
- `send({node, Node}, Name, Msg) -> Pid | {badarg, Msg} | {error, ErrorMsg}`
  [page 167] Send `Msg` to a registered process `Name`
- `send({group, GlobalGroupName}, Name, Msg) -> Pid | {badarg, Msg} | {error, ErrorMsg}`
  [page 167] Send `Msg` to a registered process `Name`
- `sync() -> ok`
  [page 167] Synchronize the immediate global group
- `whereis_name(Name) -> Pid | undefined | {error, ErrorMsg}`
  [page 168] Return the Pid of the global process `Name`
- `whereis_name({node, Node}, Name) -> Pid | undefined | {error, ErrorMsg}`
  [page 168] Return the Pid of the global process `Name`
- `whereis_name({group, GlobalGroupName}, Name) -> Pid | undefined | {error, ErrorMsg}`
  [page 168] Return the Pid of the global process `Name`
- `start()`
  [page 168] Start the global group server
- `start_link() -> {ok, Pid} | {error, Reason}`
  [page 168] Start the global group server
- `stop() -> void()`
  [page 168] Stop the global group server

heart

The following functions are exported:

- `start() -> {ok, Pid} | ignore | {error, What}`
  [page 169] Start the heart program.
- `set_cmd(Cmd) -> ok | {error, {bad_cmd, Cmd}}`
  [page 170] Set a temporary reboot command.
- `clear_cmd() -> ok`
  [page 170] Clear the temporary boot command.
- `get_cmd() -> {ok, Cmd}`
  [page 170] Get the temporary reboot command.

inet

The following functions are exported:

- `get_rc()`
  [page 172] Return list of configuration parameters.
- `format_error(Tag)`
  [page 172] Return a diagnostic error string.
- `gethostbyaddr(Address) -> {ok, Hostent} | {error, Reason}`
  [page 172] Return a `hostent` record for the host with the given address
- `gethostbyname(Name) -> {ok, Hostent} | {error, Reason}`
  [page 172] Return a `hostent` record for the host with the given name
- `gethostbyname(Name, Family) -> {ok, Hostent} | {error, Reason}`
  [page 172] Return a `hostent` record for the host with the given name
- `gethostname() -> {ok, Name} | {error, Reason}`
  [page 172] Return the local hostname
- `sockname(Socket) -> {ok, {IP, Port}} | {error, Reason}`
  [page 172] Return the local address and port number for a socket.
- `peername(Socket) -> {ok, {Address, Port}} | {error, Reason}`
  [page 173] Return the address and port for the other end of a connection.
- `port(Socket) -> {ok, Number}`
  [page 173] Return the local port number for a socket.
- `close(Socket) -> ok`
  [page 173] Close a socket of any type
- `getaddr(IP,inet) -> {ok,{A1,A2,A3,A4}} | {error, Reason}`
  [page 173] Return the IP-adress for `IP`
- `setopts(Socket, Options) -> ok | {error, Reason}`
  [page 173] Set one or more options for a socket.

init

The following functions are exported:

- `boot(BootArgs) -> void()`
  [page 179] Start the Erlang runtime system.
- `get_arguments() -> Flags`
  [page 179] Get all flag arguments.
- `get_argument(Flag) -> {ok, Values} | error`
  [page 180] Get values associated with an argument.
- `get_args() -> [Arg]`
  [page 180] Get all (non-flag) arguments.
- `get_plain_arguments() -> [Arg]`
  [page 180] Get all (non-flag) arguments.
- `restart() -> void()`
  [page 180] Restart the running Erlang node
- `reboot() -> void()`
  [page 180] Take down an Erlang node smoothly

- `stop() -> void()`
  [page 180] Take down an Erlang node smoothly<
- `get_status() -> {InternalStatus, ProvidedStatus}`
  [page 181] Get status information during system start.
- `script_id() -> Id`
  [page 181] Get the identity of the used boot script.

net_adm

The following functions are exported:

- `host_file()`
  [page 184] Read the `.hosts.erlang` file
- `dns_hostname(Host)`
  [page 184] Call epmd for the fully qualified name (DNS) of `Host`
- `localhost()`
  [page 184] Return the fully qualified name of the local host
- `names(), names(Host)`
  [page 184] Return `{ok, List}` or `{error, Reason}`
- `ping(Node)`
  [page 184] Set up a connection to a node
- `world (), world (verbose)`
  [page 184] Run `epmd - names` on all hosts which are specified in the Erlang host file
- `world_list (Hostlist), world_list (Hostlist, verbose)`
  [page 184] Run `epmd - names` on all hosts which are specified in the Erlang host file

net_kernel

The following functions are exported:

- `monitor_nodes(Flag, OptionList) -> ok | ignored | Error`
  [page 186] Subscribe/unsubscribe to nodeup and nodedown messages
- `monitor_nodes(Flag) -> ok | ignored | Error`
  [page 187] Subscribe/unsubscribe to nodeup and nodedown messages
- `allow(NodeList)`
  [page 188] Limit access to a node from a specific number of named nodes
- `connect_node(Node)`
  [page 188] Establish a connection to a node
- `set_net_ticktime(NetTicktime, TransitionPeriod) -> Res`
  [page 188] Set `net_ticktime`
- `set_net_ticktime(NetTicktime) -> Res`
  [page 189] Set `net_ticktime`
- `get_net_ticktime() -> Res`
  [page 189] Get `net_ticktime`

os

The following functions are exported:

- `cmd(Command) -> string()`
  [page 190] Execute `Command` in a command shell of the target OS.
- `find_executable(Name) -> Filename | false`
  [page 190] Return the absolute filename of a program.
- `find_executable(Name, Path) -> Filename | false`
  [page 190] Return the absolute filename of a program.
- `getenv() -> List`
  [page 190] Return a list of all environment variables.
- `getenv(VarName) -> Value | false`
  [page 191] Return the `Value` of the environment variable `VarName`.
- `getpid() -> Value`
  [page 191] Return the process identifier of the emulator process as a string.
- `putenv(VarName, Value) -> true`
  [page 191] Set a new `Value` for the environment variable `VarName`.
- `type() -> {Osfamily,Osname} | Osfamily`
  [page 191] Return the `Osfamily` and, in some cases, `Osname` of the current operating system.
- `version() -> {Major, Minor, Release} | VersionString`
  [page 191] Return the Operating System version.

packages

The following functions are exported:

- `no functions exported`
  [page 195] x

pg2

The following functions are exported:

- `create(Name) -> void()`
  [page 196] Create a new, empty process group
- `delete(Name) -> void()`
  [page 196] Delete a process group
- `get_closest_pid(Name) -> Pid | {error, Reason}`
  [page 196] Common dispatch function
- `get_members(Name) -> [Pid] | {error, Reason}`
  [page 197] Return all processes in a group
- `get_local_members(Name) -> [Pid] | {error, Reason}`
  [page 197] Return all local processes in a group
- `join(Name, Pid) -> ok | {error, Reason}`
  [page 197] Join a process to a group
- `leave(Name, Pid) -> ok | {error, Reason}`
  [page 197] Make a process leave a group

- `which_groups() -> [Name]`
  [page 197] Return a list of all known groups
- `start()`
  [page 197] Start the pg2 server
- `start_link() -> {ok, Pid} | {error, Reason}`
  [page 197] Start the pg2 server

rpc

The following functions are exported:

- `start()`
  [page 198] Start the `rpc` server
- `stop()`
  [page 198] Stop the `rpc` server
- `call(Node, Module, Function, Args)`
  [page 198] Evaluate a function call on a node
- `call(Node, Module, Function, Args, Timeout)`
  [page 198] Evaluate a function call on a node
- `cast(Node, Module, Function, Args)`
  [page 198] Run a function on a node ignoring the result
- `block_call(Node, Mod, Fun, Args)`
  [page 199] Evaluate a function call on a node in the RPC server's context
- `block_call(Node, Module, Function, Args, Timeout)`
  [page 199] Evaluate a function call on a node in the RPC server's context
- `server_call(Node, Name, ReplyWrapper, Msg)`
  [page 199] Interact with a server on a node
- `abcast(Name, Mess)`
  [page 199] Broadcast a message asynchronously to a registered process on all nodes
- `abcast(Nodes, Name, Mess)`
  [page 199] Broadcast a message asynchronously to a registered process on specific nodes
- `sbcast(Name, Msg)`
  [page 199] Broadcast to all nodes synchronously and return a list of the nodes which have a registered server
- `sbcast(Nodes, Name, Msg)`
  [page 199] Broadcast to specific nodes synchronously and return a list of the nodes which have a registered server
- `eval_everywhere(Mod, Fun, Args)`
  [page 199] Run a function call on all nodes
- `eval_everywhere(Nodes, Mod, Fun, Args)`
  [page 200] Run a function call on a specified set of nodes
- `multicall(M, F, A)`
  [page 200] Evaluate a function call on all nodes
- `multicall(Nodes, M, F, A)`
  [page 200] Evaluate a function call on a set of nodes

- multicall(M, F, A, Timeout)
  [page 200] Evaluate a function call on all nodes, with timeout
- multicall(Nodes, M, F, A, Timeout)
  [page 200] Evaluate a function call on a set of nodes, with timeout
- multi_server_call(Name, Msg)
  [page 200] Send a message to servers on all nodes and collect the answers
- multi_server_call(Nodes, Name, Msg)
  [page 201] Send a message to servers on a specific set of nodes and collect the answers
- safe_multi_server_call(Name, Msg)
  [page 201] Send a message to servers on all nodes and collect the answers safely
- safe_multi_server_call(Nodes, Name, Msg)
  [page 201] >Send a message to servers on a specific set of nodes and collect the answers safely
- async_call(Node, Mod, Fun, Args)
  [page 201] Evaluate a function asyncronously on a node and return a key which can be used at a later stage to collect results
- yield(Key)
  [page 201] Deliver the promised answer from a async_call operation
- nb_yield(Key, Timeout)
  [page 201] Deliver the promised answer from a async_call operation (non-blocking)
- nb_yield(Key)
  [page 202] Deliver the promised answer from a async_call operation (non-blocking)
- parallel_eval(ListOfTuples)
  [page 202] Evaluate several function calls on all nodes in parallel
- pmap({M, F}, Extraargs, List)
  [page 202] Execute lists:map/3 in parallel on all nodes
- pinfo(Pid)
  [page 202] Get process info for any process
- pinfo(Pid, Item)
  [page 202] Get process info for any process

seq_trace

The following functions are exported:

- set_token(Component, ComponentValue) -> {Component, PreviousValue}
  [page 203] Set the individual Component of the trace token.
- set_token(Token) -> PreviousToken
  [page 204] Set the trace token to Value.
- get_token(Component) -> {Component, ComponentValue}
  [page 204] Return the ComponentValue of the trace token component Component.
- get_token() -> TraceToken
  [page 204] Return the value of the trace token.
- print(TraceInfo) -> void
  [page 204] Put the Erlang term TraceInfo into the sequential trace output.

- `print(Label, TraceInfo) -> void`
  [page 205] Put the Erlang term `TraceInfo` into the sequential trace output.
- `reset_trace() -> void`
  [page 205] Stop all sequential tracing on the Erlang node.
- `set_system_tracer(ProcessOrPortId) -> PreviousId`
  [page 205] Set the system tracer.
- `get_system_tracer() -> pid() | port() | false`
  [page 205] Return the pid() or port() of the current system tracer.

user

The following functions are exported:

- `start() -> void()`
  [page 211] Start the standard I/O system.

wrap_log_reader

The following functions are exported:

- `chunk(Continuation)`
  [page 212] Read a chunk of objects written to a wrap log.
- `chunk(Continuation, N) -> {Continuation2, Terms} | {Continuation2, Terms, Badbytes} | {Continuation2, eof} | {error, Reason}`
  [page 212] Read a chunk of objects written to a wrap log.
- `close(Continuation) -> ok`
  [page 213] Close a log
- `open(Filename) -> OpenRet`
  [page 213] Open a log file
- `open(Filename, N) -> OpenRet`
  [page 213] Open a log file

app

No functions are exported.

config

No functions are exported.

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 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] <optional> 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 <optional> 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] <optional> 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 <optional> Value is one of:

tty All standard error reports are written to stdio. This is the default option.

{file, FileName} All standard error reports are written to the file FileName, where FileName is a string.

false No error logger handler is installed.

global_groups = [GroupTuple] <optional> Specifies the groups of nodes which will have their own global name space. In this parameter:

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

These parameters are described in global_group(3).

`inet_parse_error_log = LogMode <optional>` LogMode is one of:

`silent` No `error_logger` messages are generated when erroneous lines are found and skipped in the various configuration files. The default if the variable is not set is that erroneous lines are reported via the `error_logger`.

`net_setuptime = SetupTime <optional>` Specifies the `net_kernel` setup time. `SetupTime` is given in seconds. This is the maximum time a node will wait for the other node to answer during connection setup and handshake. If this parameter is undefined, it defaults to 7 seconds.

`net_ticktime = TickTime <optional>` 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] <optional>` 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] <optional>` 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 <optional>` 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_ddll = true | false <optional>` Starts the `ddll_server` if the parameter is `true` (see `erl_ddll(3)`). This parameter should be set to `true` an embedded system which uses this service.

The default value is `false`.

`start_dist_ac = true | false <optional>` 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 <optional>` 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] <optional> 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 <optional> 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 <optional> 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 <optional> 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.

keep_zombies = integer() <optional> Sets the value of the system flag keep_zombies.

The default value is 0.

shutdown_func = {Mod,Func} <optional> 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) [page 214], application(3) [page 33], code(3) [page 44], disk_log(3) [page 52], erl_boot_server(3) [page 66], erl_ddll(3) [page 68], error_logger(3) [page 129], file(3) [page 135], global(3) [page 160], global_group(3) [page 165], heart(3) [page 169], inet(3) [page 171], net_kernel(3) [page 186], os(3) [page 190], pg2(3) [page 196], rpc(3) [page 198], seq_trace(3) [page 203], user(3) [page 211]

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.

Exports

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

Types:

- Application = atom()
- Env = [{Par,Val}]
- 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) -> {ok, Keys} | undefined
```

Types:

- Application = atom()
- Keys = [{Key,Val}]
- 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() -> {ok, Application} | undefined
get_application(Pid | Module) -> {ok, Application} | undefined
```

Types:

- Pid = pid()
- Module = atom()
- 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) -> {ok, Val} | undefined
get_env(Application, Par) -> {ok, Val} | undefined
```

Types:

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

Types:

- Application = atom()
- 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 = atom()
- AppSpec = {application,Application,AppSpecKeys}
- AppSpec = [{Key,Val}]
- Key = atom()
- Val = term()
- Distributed = {Application,Nodes} | {Application,Time,Nodes} | default
- Nodes = [node() | {node(),..,node()}]
- Time = integer() > 0
- Reason = 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 = string()
- 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, Bool) -> ok | {error, Reason}
```

Types:

- Application = atom()
- Bool = bool()
- 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
```

Types:

- Application = atom()
- Par = atom()
- Val = term()

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

> **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 = permanent | transient | temporary
- 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 | local | undefined
```

Types:

- StartType = normal | {takeover,Node} | {failover,Node}
- Node = node()

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 = permanent | transient | temporary
- 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

Types:

- Application = atom()
- Par = atom()

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

> **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}]
```

Types:

- Application = atom()
- Description = string()
- 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 specfication keys, respectively.

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 its 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 = normal | {takeover,Node} | {failover,Node}
- Node = node()
- 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 appliction) 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) [page 29], app(4) [page 214]

auth

Erlang Module

Authentication determines which nodes are allowed to communicate with each other. In a network of different Erlang nodes, it is built into the system at the lowest possible level. Each node has its `Magic Cookie`, which is an Erlang atom.

When nodes connect with eachother, the `Magic Cookie`s are compared. If the `Magic Cookie`s doesn't match the connected node rejects the connection.

At start-up, the first action of the standard `auth` server is to read a file named `$HOME/erlang.cookie`. An atom is created from the contents of this file and the cookie of the node is set to this atom with the use of `erlang:set_cookie(node(), CookieAtom)`.

If the file does not exist, it is created. The UNIX permissions mode of the file is set to octal 400 (read-only by owner) and filled with a random string. For this reason, the same user, or group of users with identical cookie files, can have Erlang nodes which can communicate freely and without interference from the `Magic Cookie` system. Users who want to run nodes on separate file systems must be certain that their cookie files are identical on the different file systems.

Initially, each node has a random atom assigned as its magic cookie. Once the procedure described above has been concluded, the cookie is set to the contents of the `$HOME/erlang.cookie` file.

To communicate with another node, the magic cookie of that node must be known. The BIF `erlang:set_cookie(Node, Cookie)` sets the cookie for `Node` to `Cookie`. The call `erlang:set_cookie(node(), CookieAtom)` will set the current cookie to `CookieAtom`. It will, however, also set the cookie of all other unknown nodes to `CookieAtom`. In the case of the default `auth` server, this is the first thing done when the system starts. The default then, is to assume that `all` nodes which communicate have the same cookie. In the case of a single user on a single file system, this is indeed true and no further action is required. The original cookie can also be fetched by the BIF `erlang:get_cookie()`.

If nodes which communicate do not have the same cookie, they can be set explicitly on each node with the aid of `erlang:set_cookie(Node, Cookie)`. Distributed systems with multiple User IDs can be handled in this way.

Initially, the system cookie is set to a random atom, and the (assumed) cookie of all other nodes is initially set to the atom `nocookie`. Thus, an Erlang node is completely unprotected when `erlang:set_cookie(node(), nocookie)` is run. Sometimes, this may be appropriate for systems which are not normally networked, and it can also be appropriate for maintenance purposes.

In the standard system, the default when two nodes are connected is to immediately connect all other involved nodes as well. This way, there is always a fully connected network. If there are nodes with different cookies, this method might be inappropriate and the host OS command line option `-connect_all false` must be issued to the Erlang runtime system. See `global(3)`.

This module uses the two BIFs erlang:get_cookie() which returns the magic cookie of the local node, and erlang:set_cookie(Node,Cookie) which sets the magic cookie of Node to Cookie. If Node is the user's node, the cookie of all other unknown nodes are also set to Cookie by this BIF.

Exports

start()

Starts the auth server.

stop()

Stops the auth server.

is_auth(Node)

Returns the value yes if communication with Node is authorized, no if Node does not exist or communication is not authorized.

exists(Node)

Returns yes if Node exists, otherwise no.

cookie()

Reads cookie from $HOME/.erlang.cookie and sets it. This function is used by the auth server at start-up.

node_cookie(Node, Cookie)

If the cookie of Node is known to the user as Cookie but the user's cookie is not known at Node, this function informs Node of the identity of the user's cookie.

node_cookie([Node, Cookie])

Another version of the previous function with the arguments in a list which can be given on the host OS command line.

cookie([Cookie])

Equivalent to erlang:set_cookie(node(), Cookie), but with the argument in a list so it can be given on the host OS command line.

code

Erlang Module

This module deals with the loading of compiled code into a running Erlang runtime system.

The code server dynamically loads modules into the system on demand, which means the first time the module is referenced. This functionality can be turned off using the command line flag `-mode embedded`. In this mode, all code is loaded during system start-up.

If started in `interactive` mode, all directories under the $ROOT/lib directory are initially added to the search path of the code server (). The $ROOT directory is the installation directory of Erlang/OTP, `code:root_dir()`. Directories can be named `Name[-Vsn]` and the code server, by default, chooses the greatest (>) directory among those which have the same `Name`. The `-Vsn` suffix is optional.

If an `ebin` directory exists under a chosen directory, it is added to the directory. The `Name` of the directory (or library) can be used to find the full directory name (including the current version) through the `priv_dir/1` and `lib_dir/1` functions.

The code server incorporates a code path cache. The cache functionality is disabled by default. To activate it, start the emulator with 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 cache. This feature is used by `application` 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.

Exports

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

Types:

- Flags = [stick | nostick | embedded | interactive]
- Pid = pid()
- What = term()

This function starts the code server. `start/0` implies that the `stick` and `interactive` flags are set.

`Flags` can also be entered as the command line flags `-stick`, `-nostick` and `-mode embedded | interactive`. `-stick` and `-mode interactive` are the defaults. The `stick` flag indicates that a module can never be re-loaded once it has been loaded from the `kernel`, `stdlib`, or `compiler` directories.

```
start_link() -> {ok, Pid} | {error, What}
start_link(Flags) -> {ok, Pid} | {error, What}
```

Types:

- Flags = [stick | nostick | embedded | interactive]
- Pid = pid()
- What = term()

This function starts the code server and sets up a link to the calling process. This function should be used if the code server is supervised. `start_link/0` implies that the `stick` and `interactive` flags are set.

The `Flags` can also be given as command line flags, `-stick`, `-nostick` and `-mode embedded | interactive` where `-stick` and `-mode interactive` is the default. The `stick` flag indicates that a module which has been loaded from the `kernel`, `stdlib` or `compiler` directories can never be reloaded.

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

Types:

- DirList = [Dir]
- Dir = string()
- What = bad_directory | bad_path

Sets the code server search path to the list of directories `DirList`.

```
get_path() -> Path
```

Types:

- Path = [Dir]
- Dir = string()

Returns the current path.

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

Types:

- Dir = string()
- What = bad_directory

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

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

Types:

- Dir = string()
- What = bad_directory

This function adds `Dir` to the beginning of the current path. If `Dir` already exists, the old directory is removed from path.

`add_paths(DirList) -> ok`

`add_pathsz(DirList) -> ok`

Types:

- DirList = [Dir]
- Dir = string()

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

`add_pathsa(DirList) -> ok`

Types:

- DirList = [Dir]
- Dir = string()

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

`del_path(NameDir) -> true | false | {error, What}`

Types:

- NameDir = Name | Dir
- Name = atom()
- Dir = string()
- What = bad_name

This function deletes an old occurrence of a directory in the current path with the name `.../Name[-*][/ebin]`. It is also possible to give the complete directory name `Dir` in order to delete it.

This function returns `true` if the directory was deleted, and `false` if the directory was not found.

`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[-*][/ebin]`, in the current path, with `Dir`. If `Name` does not exist, it adds the new directory `Dir` last in path. The new directory must also be named `.../Name[-*][/ebin]`. This function should be used if a new version of the directory (library) is added to a running system.

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

Types:

- Module = atom()
- What = nofile | sticky_directory | badarg | term()

This function tries to load the Erlang module `Module`, using the current path. It looks for the object code file which has a suffix 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.

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

Types:

- File = atom() | string()
- Module = atom()
- What = nofile | sticky_directory | badarg | term()

This function does the same as `load_file(Module)`, but `File` is either an absolute file name, or a relative file name. The current path is not searched. It returns a value in the same way as `load_file(Module)`. Note that `File` should not contain an extension (`".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()

This function tries to ensure that the module `Module` is loaded. To work correctly, a file with the same name as `Module.Suffix` must exist in the current search path. `Suffix` must correspond to the running Erlang machine, for example `.beam`. It returns a value in the same way as `load_file(File)`.

If the system is started with the `-mode embedded` command line flag, this function will not load a module which has not already been loaded. `{error, embedded}` is returned.

`delete(Module) -> true | false`

Types:

- Module = atom()

This function deletes the code in `Module` and the code in `Module` is marked as old. This means that no external function calls can be made to this occurrence of `Module`, but a process which executes code inside this module continues to do so. Returns `true` if the operation was successful (i.e., there was a current version of the module, but no old version), otherwise `false`.

`purge(Module) -> true | false`

Types:

- Module = atom()

This function purges the code in `Module`, that is, it removes code marked as old. If some processes still execute code in the old occurrence of `Module`, these processes are killed before the module is purged. Returns `true` if a process has been killed, otherwise `false`.

`soft_purge(Module) -> true | false`

Types:

- Module = atom()

This function purges the code in `Module`, that is, it removes code marked as old, but only if no process currently runs the old code. It returns `false` if a process uses the old code, otherwise `true`.

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

Types:

- Module = atom()
- Loaded = AbsFileName | preloaded
- AbsFileName = string()

This function tests if module `Module` is loaded. If the module is loaded, the absolute file name of the file from which the code was obtained is returned.

`all_loaded() -> [LoadMod]`

Types:

- LoadMod = {Module, Loaded}
- Module = atom()
- Loaded = AbsFileName | preloaded
- AbsFileName = string()

This function returns a list of tuples of the type {`Module, Loaded`} for all loaded modules. `Loaded` is the absolute file name of the loaded module, or the atom `preloaded` if the module was pre-loaded.

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

Types:

- Module = atom()
- What = sticky_directory | badarg | term()

This function can be used to load object code on remote Erlang nodes. It can also be used to load object code where the file name and module name differ. This, however, is a very unusual situation and should be used with care. The parameter `Binary` must contain object code for the module `Module`. The `File` parameter is only used by the code server to keep a record from which file the object code in `Module` comes. Accordingly, `File` is not opened and read by the code server.

`stop() -> stopped`

Stops the code server.

`root_dir() -> RootDir`

Types:

- RootDir = string()

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

`lib_dir() -> LibDir`

Types:

- LibDir = string()

Returns the library directory.

`lib_dir(Name) -> LibDir | {error, What}`

Types:

- Name = atom()
- LibDir = string()
- What = bad_name

This function returns the current `lib` directory for the `Name[-*]` directory (or library). The current path is searched for a directory named `.../Name-*` (the `-*` suffix is optional for directories in the search path and it represents the version of the directory).

`compiler_dir() -> CompDir`

Types:

- CompDir = string()

This function returns the compiler directory.

`priv_dir(Name) -> PrivDir | {error, What}`

Types:

- Name = atom()
- PrivDir = string()
- What = bad_name

This function returns the current `priv` directory for the `Name[-*]` directory. The current path is searched for a directory named `.../Name-*` (the `-*` suffix is optional for directories in the search path and it represents the version of the directory). The `/priv` suffix is added to the end of the found directory.

`get_object_code(Module) -> {Module, Bin, AbsFileName} | error`

Types:

- Module = atom()
- Bin = binary()
- AbsFileName = string()

This function searches the code path in the code server for the object code of the module `Module`. It returns `{Mod, Bin, Filename}` if successful, and `error` if not. `Bin` 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 node `N` is done as follows:

```
...
{Mod, B, F} = code:get_object_code(Mod),
rpc:call(N,code, load_binary, [Mod, F, B]),
...
```

`objfile_extension() -> Ext`

Types:

- Ext = string()

This function returns the object code file extension for the running Erlang machine, for example ".beam".

`rehash() -> ok`

This function creates or rehashes the code path cache.

`stick_dir(Dir) -> ok | {error, term()}`

Types:

- Dir = string()

This function marks `Dir` as 'sticky'. The system issues a warning and rejects the request if a user tries to re-load a module in a sticky directory. Sticky directories are used to warn the user about inadvertent changes to system software.

`unstick_dir(Dir) -> ok | {error, term()}`

Types:

- Dir = string()

This function unsticks a directory which has been marked sticky. Code which is located in the unstuck directory can be re-loaded into the system.

`which(Module) -> WhichFile`

Types:

- Module = atom()
- WhichFile = FileName | non_existing | preloaded | cover_compiled
- FileName = string()

If the module is not loaded already, this function returns the directory path to the first file name in the search path of the code server which contains the object code for `Module`. If the module is loaded, it returns the directory path to the file name which contains 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.

`where_is_file(File) -> FullName`

Types:

- File = string()
- FullName = string() | non_existing

This function searches the directories in the code path for `File` (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 e.g. to locate application resource files. If the code path cache is used, the code server will efficiently read the full name from the cache (if `File` is an object code file or a `.app` file, that is).

`clash() -> ok`

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

Notes

`Dir` has the described type `string()` in all functions. For backwards compatibility, `atom()` is also allowed, but `string()` is recommended.

The described type for `Module` is `atom()` in all functions. For backwards compatibility, `string()` is also allowed.

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 [page 62]. 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.

Exports

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

Types:

- LocalLog = DistributedLog = term()

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)
balog(Log, Bytes) -> ok | {error, Reason}
```

Types:

- Log = term()
- Term = term()
- Bytes = binary() | [Byte]
- Byte = [Byte] | 0 =< integer() =< 255
- Reason = 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)
balog_terms(Log, BytesList) -> ok | {error, Reason}
```

Types:

- Log = term()
- TermList = [term()]
- BytesList = [Bytes]
- Bytes = binary() | [Byte]
- Byte = [Byte] | 0 =< integer() =< 255
- Reason = 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)
block(Log, QueueLogRecords) -> ok | {error, Reason}
```

Types:

- Log = term()
- QueueLogRecords = bool()
- Reason = 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 = term()
- Header = {head, Head} | {head_func, {M,F,A}}
- Head = none | term() | binary() | [Byte]
- Byte = [Byte] | 0 =< integer() =< 255
- 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 = term()
- Owner = pid()
- Notify = bool()
- 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 = term()
- Size = integer() > 0 | infinity | {MaxNoBytes, MaxNoFiles}
- MaxNoBytes = integer() > 0
- MaxNoFiles = integer() > 0

- Reason = no_such_log | nonode | {read_only_mode, Log} | {blocked_log, Log} | {new_size_too_small, CurrentSize} | {badarg, size} | {file_error, FileName, FileError}

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(Log, Continuation, N) -> {Continuation2, Terms} | {Continuation2, Terms,
            Badbytes} | eof | {error, Reason}
bchunk(Log, Continuation)
bchunk(Log, Continuation, N) -> {Continuation2, Binaries} | {Continuation2, Binaries,
            Badbytes} | eof | {error, Reason}
```

Types:

- Log = term()
- Continuation = start | cont()
- N = integer() > 0 | infinity
- Continuation2 = cont()
- Terms = [term()]
- Badbytes = integer()
- Reason = no_such_log | {format_external, Log} | {blocked_log, Log} | {badarg, continuation} | {not_internal_wrap, Log} | {corrupt_log_file, FileName} | {file_error, FileName, FileError}
- Binaries = [binary()]

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 = cont()
- 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, Continuation2} | {error, Reason}

Types:

- Log = term()
- Continuation = start | cont()
- Step = integer()
- Continuation2 = cont()
- Reason = no_such_log | end_of_log | {format_external, Log} | {blocked_log, Log} | {badarg, continuation} | {file_error, FileName, FileError}

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, Reason}

Types:

- Reason = no_such_log | nonode | {file_error, FileName, FileError}

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) -> character_list()

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, Reason}

Types:

- Reason = no_such_log | nonode | {read_only_mode, Log} | {blocked_log, Log} | {halt_log, Log} | {invalid_header, InvalidHeader} | {file_error, FileName, FileError}

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}

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 [page 62].
- {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)
lclose(Log, Node) -> ok | {error, Reason}
```

Types:

- Node = node()
- Reason = no_such_log | {file_error, FileName, FileError}

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 [page 58].

If there is no log with the given name on the specified node, `no_such_log` is returned.

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

Types:

- Log = term()
- Term = term()
- Bytes = binary() | [Byte]
- Byte = [Byte] | 0 =< integer() =< 255
- Reason = no_such_log | nonode | {read_only_mode, Log} | {format_external, Log} | {blocked_log, Log} | {full, Log} | {invalid_header, InvalidHeader} | {file_error, FileName, FileError}

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)
blog_terms(Log, BytesList) -> ok | {error, Reason}
```

Types:

- Log = term()
- TermList = [term()]
- BytesList = [Bytes]
- Bytes = binary() | [Byte]
- Byte = [Byte] | 0 =< integer() =< 255
- Reason = no_such_log | nonode | {read_only_mode, Log} | {format_external, Log} | {blocked_log, Log} | {full, Log} | {invalid_header, InvalidHeader} | {file_error, FileName, FileError}

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) -> OpenRet | DistOpenRet`

Types:

- ArgL = [Opt]
- Opt = {name, term()} | {file, FileName}, {linkto, LinkTo} | {repair, Repair} | {type, Type} | {format, Format} | {size, Size} | {distributed, [Node]} | {notify, bool()} | {head, Head} | {head_func, {M,F,A}} | {mode, Mode}
- FileName = string() | atom()
- LinkTo = pid() | none
- Repair = true | false | truncate
- Type = halt | wrap
- Format = internal | external
- Size = integer() > 0 | infinity | {MaxNoBytes, MaxNoFiles}
- MaxNoBytes = integer() > 0
- MaxNoFiles = 0 < integer() < 65000
- Rec = integer()
- Bad = integer()
- Head = none | term() | binary() | [Byte]
- Byte = [Byte] | 0 =< integer() =< 255
- Mode = read_write | read_only
- OpenRet = Ret | {error, Reason}
- DistOpenRet = {[{Node, Ret}], [{BadNode, {error, DistReason}}]}
- Node = BadNode = atom()
- Ret = {ok, Log} | {repaired, Log, {recovered, Rec}, {badbytes, Bad}}
- DistReason = nodedown | Reason
- Reason = no_such_log | {badarg, Arg} | {size_mismatch, CurrentSize, NewSize} | {arg_mismatch, OptionName, CurrentValue, Value} | {name_already_open, Log} | {open_read_write, Log} | {open_read_only, Log} | {need_repair, Log} | {not_a_log_file, FileName} | {invalid_index_file, FileName} | {invalid_header, InvalidHeader} | {file_error, FileName, FileError} | {node_already_open, Log}

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, Name}} 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:

- Log = term()
- Pid = pid()

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

Types:

- Log = term()
- File = string()
- Head = term()
- BHead = binary() | [Byte]
- Byte = [Byte] | 0 =< integer() =< 255
- Reason = no_such_log | nonode | {read_only_mode, Log} | {blocked_log, Log} | {same_file_name, Log} | {invalid_index_file, FileName} | {invalid_header, InvalidHeader} | {file_error, FileName, FileError}

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, Reason}
```

Types:

- Log = term()
- Reason = no_such_log | nonode | {read_only_mode, Log} | {blocked_log, Log} | {file_error, FileName, FileError}

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)
truncate(Log, Head)
btruncate(Log, BHead) -> ok | {error, Reason}
```

Types:

- Log = term()
- Head = term()
- BHead = binary() | [Byte]
- Byte = [Byte] | 0 =< integer() =< 255
- Reason = no_such_log | nonode | {read_only_mode, Log} | {blocked_log, Log} | {invalid_header, InvalidHeader} | {file_error, FileName, FileError}

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, Reason}
```

Types:

- Log = term()
- Reason = 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) [page 135], pg2(3) [page 196], wrap_log_reader(3) [page 212]

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` flag must have one instance of this server running.

This server can be started with the `kernel` configuration parameter `start_boot_server`.

Exports

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

Types:

- Slaves = [Host]
- Host = atom()
- Pid = pid()
- What = void()

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 = void()

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 = void()

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

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

Types:

- Slave = Host
- Host = atom()
- What = void()

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) [page 179], erl_prim_loader(3) [page 71]

erl_ddll

Erlang Module

The erl_ddll module can load and link a linked-in driver, if run-time loading and linking of shared objects, or dynamic libraries, is supported by the underlying operating system.

Exports

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

Starts ddll_server. The error return values are the same as for gen_server.

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

Starts ddll_server and links it to the calling process. The error return values are the same as for gen_server.

stop() -> ok

Stops ddll_server.

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

Types:

- Name = string() | atom()
- Path = string() | atom()

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 Paths cannot be loaded under the same name. The number of dynamically loadable drivers are limited by the size of driver_tab in config.c.

If the server is not started the caller will crash.

unload_driver(Name) -> ok | {error, ErrorDescriptor}

Types:

- Name = string() | atom()

Unloads the dynamic driver `Name`. This will fail if any port programs are running the code that is being unloaded. Linked-in drivers cannot be unloaded. The process must previously have called `load_driver/1` for the driver.

There is no guarantee that the memory where the driver was loaded is freed. This depends on the underlying operating system.

If the server is not started the caller will crash.

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

Types:

- DriverList = [Driver()]
- Driver = string()

Returns a list of all the available drivers, both (statically) linked-in and dynamically loaded ones.

If the server is not started the caller will crash.

`format_error(ErrorDescriptor) -> string()`

Takes an `ErrorDescriptor` which has been returned by one of `load_driver/2` and `unload_driver/1` and returns a string which describes the error or warning.

Differences Between Statically Linked-in Drivers and Dynamically Loaded Drivers

Except for the following minor changes, all information in Appendix E of Concurrent Programming in Erlang, second edition, still applies.

Before the driver is unloaded, the `finish` function is called, without arguments, to give the driver writer a chance to clean up and release memory allocated in `driver_init`.

After the driver is loaded, the function `struct driver_entry *driver_init(void *)` is called with `handle` as argument. If the operating system loader cannot find a function called `driver_init`, the driver will not be loaded. The `driver_init` function *must* initialize a `ErlDrvEntry` struct and return a pointer to it.

The name of the driver, returned from `driver_init` must match the name of the driver, and the file name (with extensions removed).

Example:

```
#include <stdio.h>
#include "erl_driver.h"
static long my_start(ErlDrvPort, char*);
static int my_stop(ErlDrvData), my_read(ErlDrvData, char*, int);
static ErlDrvEntry my_driver_entry;
/*
 * Initialize and return a driver entry struct
 */
ErlDrvEntry *driver_init(void *handle)
{
    memset(my_driver_entry, '\0', sizeof(my_driver_entry));
    my_driver_entry.start = my_start;
    my_driver_entry.stop = my_stop;
```

```
    my_driver_entry.output = my_read;
    my_driver_entry.driver_name = "my_driver";
    my_driver_entry.finish = null_func;
    return &my_driver_entry;
}
```

config.c

The size of the `driver_tab` array, defined in `config.c`, limits the number of dynamically loadable drivers.

Compiling Your Driver

Please refer to your C compiler or operating system documentation for information about producing a sharable object or DLL.

The include file `erl_driver.h` is found in the `usr/include` directory of the Erlang installation. Also the older file `driver.h` is available, but is considered obsolete.

SEE ALSO

erl_driver(4), driver_entry(4)

erl_prim_loader

Erlang Module

The erl_prim_loader is used to load all Erlang modules into the system. The start script is also fetched with the low level loader.

The erl_prim_loader knows about the environment and how to fetch modules. The loader could, for example, fetch files using the file system (with absolute file names as input), or a database (where the binary format of a module is stored).

The -loader Loader command line flag can be used to choose the method used by the erl_prim_loader. Two Loader methods are supported by the Erlang runtime system: efile and inet. If another loader is required, then it has to be implemented by the user. The Loader provided by the user must fulfill the protocol defined below, and it is started with the erl_prim_loader by evaluating open_port({spawn,Loader},[binary]).

Exports

start(Id,Loader,Hosts) -> {ok, Pid} | {error, What}

Types:

- Id = term()
- Loader = atom() | string()
- Hosts = [Host]
- Host = atom()
- Pid = pid()
- What = term()

Starts the Erlang low level loader. This function is called by the init process (and module). The init process reads the command line flags -id Id, -loader Loader, and -hosts Hosts. These are the arguments supplied to the start/3 function.

If -loader is not given, the default loader is efile which tells the system to read from the file system.

If -loader is inet, the -id Id, -hosts Hosts, and -setcookie Cookie flags must also be supplied. Hosts identifies hosts which this node can contact in order to load modules. One Erlang runtime system with a erl_boot_server process must be started on each of hosts given in Hosts in order to answer the requests. See erl_boot_server(3).

If -loader is something else, the given port program is started. The port program is supposed to follow the protocol specified below.

get_file(File) -> {ok, Bin, FullName} | error

Types:

- File = string()
- Bin = binary()
- FullName = string()

This function fetches a file using the low level loader. `File` is either an absolute file name or just the name of the file, for example `"lists.beam"`. If an internal path is set to the loader, this path is used to find the file. If a user supplied loader is used, the path can be stripped off if it is obsolete, and the loader does not use a path. `FullName` is the complete name of the fetched file. `Bin` is the contents of the file as a binary.

`get_path() -> {ok, Path}`

Types:

- Path = [Dir]
- Dir = string()

This function gets the path set in the loader. The path is set by the `init` process according to information found in the start script.

`set_path(Path) -> ok`

Types:

- Path = [Dir]
- Dir = string()

This function sets the path of the loader if `init` interprets a `path` command in the start script.

Protocol

The following protocol must be followed if a user provided loader port program is used. The `Loader` port program is started with the command `open_port({spawn,Loader},[binary])`. The protocol is as follows:

```
Function          Send               Receive
------------------------------------------------------------
get_file          [102 | FileName]   [121 | BinaryFile] (on success)
                                     [122]              (failure)

stop              eof                terminate
```

Command Line Flags

The erl_prim_loader module interprets the following flags:

-loader Loader Specifies the name of the loader used by erl_prim_loader. Loader can be efile (use the local file system), or inet (load using the boot_server on another Erlang node). If Loader is user defined, the defined Loader port program is started.

If the -loader flag is omitted, it defaults to efile.

-hosts Hosts Specifies which other Erlang nodes the inet loader can use. This flag is mandatory if the -loader inet flag is present. On each host, there must be on Erlang node with the erl_boot_server which handles the load requests. Hosts is a list of IP addresses (hostnames are not acceptable).

-id Id Specifies the identity of the Erlang runtime system. If the system runs as a distributed node, Id must be identical to the name supplied with the -sname or -name distribution flags.

-setcookie Cookie Specifies the cookie of the Erlang runtime system. This flag is mandatory if the -loader inet flag is present.

SEE ALSO

init(3) [page 179], erl_boot_server(3) [page 66]

erlang

Erlang Module

By convention, most built-in functions (BIFs) are seen as being in the module `erlang`. A number of the BIFs are viewed more or less as part of the Erlang programming language and are *auto-imported*. Thus, it is not necessary to specify the module name and both the calls `atom_to_list(Erlang)` and `erlang:atom_to_list(Erlang)` are identical.

In the text, auto-imported BIFs are listed without module prefix. BIFs listed with module prefix are not auto-imported.

BIFs may fail for a variety of reasons. All BIFs fail with reason `badarg` if they are called with arguments of an incorrect type. The other reasons that may make BIFs fail are described in connection with the description of each individual BIF.

Some BIFs may be used in guard tests, these are marked with "Allowed in guard tests".

Some BIFs, such as list_to_binary_1 [page 88], take *I/O lists* as documents (written as `iolist()` in type descriptions). An I/O list is a deep list of binaries, integers in the range 0 through 255, and other I/O lists. In an I/O list, a binary is allowed as the tail of a list.

Exports

`abs(Number)`

Returns an integer or float which is the arithmetical absolute value of the argument `Number` (integer or float).

```
> abs(-3.33).
3.33000
> abs(-3).
3
```

Allowed in guard tests.

Failure: `badarg` if `Number` is not a number.

`erlang:append_element(Tuple, Term)`

Returns a new tuple which has one element more than `Tuple`, and contains the elements in `Tuple` followed by `Term` as the last element. Semantically equivalent to `list_to_tuple(tuple_to_list(Tuple ++ [Term])`, but much faster.

Failure: `badarg` if `Tuple` is not a tuple.

`apply(Fun, ArgumentList)`

Types:

- Fun = fun()
- ArgumentList = list()

Call a fun, passing the elements in ArgumentList as arguments.

Note: If the number of elements in the arguments are known at compile-time, the call is better written as Fun(Arg1, Arg2, ... ArgN).

Fun can also be given as {Module,Function}, which is equivalent to apply(Module, Function, ArgumentList). *This usage is considered obsolete and may stop working in a future release of Erlang/OTP.*

apply(Module, Function, ArgumentList)

Returns the result of applying Function in Module to ArgumentList. The applied function must have been exported from Module. The arity of the function is the length of ArgumentList.

```
> apply(lists, reverse, [[a, b, c]]).
[c,b,a]
```

apply can be used to evaluate BIFs by using the module name erlang.

```
> apply(erlang, atom_to_list, ['Erlang']).
"Erlang"
```

Note: If the number of arguments are known at compile-time, the call is better written as Module:Function(Arg1, Arg2, ... ArgN).

Failure: error_handler:undefined_function/3 is called if Module has not exported Function/Arity. The error handler can be redefined (see the BIF process_flag/2). If the error_handler is undefined, or if the user has redefined the default error_handler so the replacement module is undefined, an error with the reason undef will be generated.

atom_to_list(Atom)

Returns a list of integers (Latin-1 codes), which corresponds to the text representation of the argument Atom.

```
> atom_to_list('Erlang').
"Erlang"
```

Failure: badarg if Atom is not an atom.

binary_to_list(Binary)

Returns a list of integers which correspond to the bytes of Binary.

binary_to_list(Binary, Start, Stop)

As binary_to_list/1, but it only returns the list from position Start to position Stop. Start and Stop are integers. Positions in the binary are numbered starting from 1.

binary_to_term(Binary)

Returns an Erlang term which is the result of decoding the binary Binary. Binary is encoded in the Erlang external binary representation. See term_to_binary/1.

erlang:bump_reductions(Reductions)

This implementation-dependent function increments the reduction counter for the current process. In the Beam emulator, the reduction counter is normally incremented by one for each function and BIF call, and a context switch is forced when the counter reaches 1000.

> Warning:
> This BIF might be removed in a future version of the Beam machine without prior warning. It is unlikely to be implemented in other Erlang implementations. If you think that you must use it, encapsulate it your own wrapper module, and/or wrap it in a catch.

erlang:cancel_timer(Ref)

cancel_timer(Ref) cancels a timer, where Ref was returned by either send_after/3 or start_timer/3. If the timer was there to be removed, cancel_timer/1 returns the time in ms left until the timer would have expired, otherwise false (which means that Ref was never a timer, or that it had already been cancelled, or that it had already delivered its message).

Note: Usually, cancelling a timer does not guarantee that the message has not already been delivered to the message queue. However, in the special case of a process P cancelling a timer which would have sent a message to P itself, attempting to read the timeout message from the queue is guaranteed to remove the timeout in that situation:

```
cancel_timer(Ref) ->
    receive
        {timeout, Ref, _} ->
            ok
    after 0 ->
        ok
    end
```

Failure: badarg if Ref is not a reference.

check_process_code(Pid, Module)

Returns true if the process Pid is executing an old version of Module, if the current call of the process executes code for an old version of the module, if the process has references to an old version of the module, or if the process contains funs that references the old version of the module. Otherwise, it returns false.

```
> check_process_code(Pid, lists).
false
```

Failure: badarg if Pid is not a pid or Module is not an atom.

`concat_binary(ListOfBinaries)`

Don't use; use list_to_binary_1 [page 88] instead.

`date()`

Returns the current date as {`Year, Month, Day`}

```
> date().
{1995, 2, 19}
```

`delete_module(Module)`

Makes the current version of the code of `Module` to the old version and deletes all export references of `Module`. Returns `undefined` if the module does not exist, otherwise `true`.

```
> delete_module(test).
true
```

Failure: `badarg` if there is already an old version of the module (see `purge_module/1`).

> **Warning:**
> This BIF is intended for code server (see `code(3)`) and should not be used elsewhere.

`erlang:demonitor(Ref)`

If `Ref` is a reference which the current process obtained by calling erlang:monitor/2 [page 93], the monitoring is turned off. No action is performed if the monitoring already is turned of before the call. Returns `true`.

After the call to `erlang:monitor/2` the monitoring process will not get any new `'DOWN'` message from this monitor into the receive queue.

It is an error if `Ref` refers to a monitoring started by another process. Not all such cases are cheap to check; if checking is cheap, the call fails with `badarg` (for example if `Ref` is a remote reference).

`disconnect_node(Node)`

Forces the disconnection of a node. This will appear to the node `Node` as if the current node has crashed. This BIF is mainly used in the Erlang network authentication protocols. Returns `true` if disconnection succeeds, otherwise `false`.

Failure: `badarg` if `Node` is not an atom.

`erlang:display(Term)`

Prints a text representation of Term on the standard output.

> **Warning:**
> This BIF is intended for debugging only.

`element(N, Tuple)`

Returns the Nth element (numbering from 1) of Tuple.

```
> element(2, {a, b, c}).
b
```

Allowed in guard tests.

Failure: badarg if N<1, or N>size(Tuple), or if Tuple is not a tuple.

`erase()`

Returns the process dictionary and deletes it.

```
> put(key1, {1, 2, 3}),
  put(key2, [a, b, c]),
  erase().
[{key1,{1,2,3}},{key2,[a,b,c]}]
```

`erase(Key)`

Returns the value associated with Key and deletes it from the process dictionary. Returns undefined if no value is associated with Key. Key can be any Erlang term.

```
> put(key1, {merry, lambs, are, playing}),
  X = erase(key1),
  {X, erase(key1)}.
{{merry,lambs,are,playing},undefined}
```

`erlang:error(Reason)`

Stops the execution of the current process with the reason Reason, where Reason is any term. The actual EXIT reason will be {Reason, Where}, where Where is a list of the functions most recently called (the current function first). Since evaluating this function causes the process to terminate, it has no return value.

`erlang:error(Reason, Args)`

Stops the execution of the current process with the reason Reason, where Reason is any term. The actual EXIT reason will be {Reason, Where}, where Where is a list of the functions most recently called (the current function first). Args is expected to be the list of arguments for the current function; in Beam it will be used to provide the actual arguments for the current function in the Where term. Since evaluating this function causes the process to terminate, it has no return value.

`exit(Reason)`

Stops the execution of the current process with the reason `Reason`. Can be caught. `Reason` is any Erlang term. Since evaluating this function causes the process to terminate, it has no return value.

```
> exit(foobar).
** exited: foobar **
> catch exit(foobar).
{'EXIT', foobar}
```

`exit(Pid, Reason)`

Sends an EXIT signal with reason `Reason` to the process `Pid`. Returns `true`.

```
> exit(Pid, goodbye).
true
```

> **Note:**
> The above is not necessarily the same as:
>
> ```
> Pid ! {'EXIT', self(), goodbye}
> ```

The above two alternatives are the same if the process with the process identity `Pid` is trapping exits. However, if `Pid` is not trapping exits, `Pid` itself will exit with reason `Reason`.

If the reason is the atom `kill`, that is if `exit(Pid, kill)` is called, an untrappable EXIT signal is sent to `Pid` which will unconditionally exit with reason `killed`.

Returns `true`.

Failure: `badarg` if `Pid` is not a pid.

`erlang:fault(Reason)`

Stops the execution of the current process with the reason `Reason`. This an old equivalent to erlang:error(Reason) [page 78].

`erlang:fault(Reason, Args)`

Stops the execution of the current process with the reason `Reason`. This an old equivalent to erlang:error(Reason, Args) [page 78].

`float(Number)`

Returns a float by converting `Number` to a float.

```
> float(55).
55.0000
```

> **Note:**
> `float/1` is also allowed in guards. But note that if it used on the top-level in a guard, it will test whether the argument is a floating point number; for clarity, use is_float/1 [page 85] instead. When `float/1` is used in an expression in a guard, such as '`float(A) == 4.0`', it behaves in the same way as in a function body.

Failure: `badarg` if `Number` is not a number.

`float_to_list(Float)`

Returns a list of integers (ASCII codes) which corresponds to `Float`.

```
> float_to_list(7.0).
"7.00000000000000000000e+00"
```

Failure: `badarg` if `Float` is not a float.

`erlang:fun_info(Fun)`

Returns a list containing information about the fun `Fun`. The list returned contains the following tuples, not necessarily in the order listed here (i.e., you should not depend on the order).

> **Warning:**
> This BIF is intended for debugging only.

{`pid`,`Pid`} `Pid` is the pid of the process that originally created the fun. It will be the atom `undefined` if the fun is given in the tuple representation.

{`module`,`Module`} `Module` (an atom) is the module in which the fun is defined.

{`name`,`Name`} `Name` is the name of the (non-exported) function that implements the fun.

{`name`,`Name`} `Name` is the name of the local function that implements the fun. If no code is currently loaded for the fun, `[]` will be returned instead of an atom.

{`arity`,`Arity`} `Arity` is the number of arguments that the fun should be called with.

{`index`,`Index`} `Index` (an integer) is an index into the module's fun table.

{`uniq`,`Uniq`} `Uniq` (an integer) is a unique value for this fun.

{`env`,`Env`} `Env` (a list) is the environment or free variables for the fun.

`erlang:fun_info(Fun, Item)`

Returns information about `Fun` as specified by `Item`, in the form {`Item`, `Info`}. `Item` can be any of the atoms `id`, `module`, `index`, `uniq`, `name`, `arity`, or `env`. See the `erlang:fun_info/1` BIF.

`erlang:fun_to_list(Fun)`

Returns a textual representation of the fun Fun.

erlang:function_exported(Module, Function, Arity)

Returns true if the module Module is loaded and contains an exported function Function/Arity; otherwise false.

Returns false for any BIF (functions implemented in C rather than in Erlang).

This function is retained mainly for backwards compatibility. It is not clear why you really would want to use it.

garbage_collect()

Forces an immediate garbage collection of the currently executing process. You should not use this function unless you have noticed or have good reasons to suspect that the spontaneous garbage collection will occur too late or not at all. Improper use may seriously degrade system performance.

Compatibility note: In versions of OTP prior to R7, the garbage collection took place at the next context switch, not immediately. To force a context switch after a call to erlang:garbage_collect(), it was sufficient to make any function call.

garbage_collect(Pid)

Works like erlang:garbage_collect() but on any process. The same caveats apply. Returns false if Pid refers to a dead process; true otherwise.

get()

Returns the process dictionary as a list of {Key, Value} tuples.

```
> put(key1, merry),
  put(key2, lambs),
  put(key3, {are, playing}),
  get().
[{key1,merry},{key2,lambs},{key3,{are,playing}}]
```

get(Key)

Returns the value associated with Key in the process dictionary, or undefined if there is no such key. Key is any term.

```
> put(key1, merry),
  put(key2, lambs),
  put({any, [valid, term]}, {are, playing}),
  get({any, [valid, term]}).
{are,playing}
```

erlang:get_cookie()

Returns the magic cookie of the current node, if the node is alive; otherwise the atom nocookie.

get_keys(Value)

Returns a list of keys which corresponds to `Value` in the process dictionary.

```
> put(mary, {1, 2}),
  put(had, {1, 2}),
  put(a, {1, 2}),
  put(little, {1, 2}),
  put(dog, {1, 3}),
  put(lamb, {1, 2}),
  get_keys({1, 2}).
[mary,had,a,little,lamb]
```

`erlang:get_stacktrace()`

Get the stacktrace of the last exception in the running process as a list of {`Module`,`Function`,`Arity`} tuples. The `Arity` field in the first tuple may be the argument list of that function instead of an arity integer, depending on the exception.

If there has not been any exceptions in a process the stacktrace is `[]`. After a code change for the process the stacktrace may also be reset to `[]`.

The stacktrace is the same data you get from the `catch` operator, for example:

```
{'EXIT',{badarg,Stacktrace}} = catch abs(x)
```

See also erlang:error/1 [page 78] and erlang:error/2 [page 78].

`group_leader()`

Returns the pid `Pid` of the group leader for the process which evaluates the function.

Every process is a member of some process group and all groups have a *group leader*. When a new process is spawned, the group leader of the spawned process is the same as that of the process which spawned it. Initially, at system start-up, `init` is both its own group leader and the group leader of all processes.

`group_leader(Leader, Pid)`

Sets the group leader of `Pid` to `Leader`. Typically, this is used when a processes started from a certain shell should have another group leader than `init`. The process `Leader` is normally a process with an I/O protocol. All I/O from this group of processes are thus channeled to the same place.

`halt()`

Halts the Erlang runtime system and indicates normal exit to the calling environment. Has no return value.

```
> halt().
os_prompt%
```

`halt(Status)`

Status must be a non-negative integer, or a string. Halts the Erlang runtime system. Has no return value. If Status is an integer, it is returned as an exit status of Erlang to the calling environment. If Status is a string, produces an Erlang crash dump with String as slogan, and then exits with a non-zero status code.

Note that on many platforms, only the status codes 0-255 are supported by the operating system.

erlang:hash(Term, Range)

Returns a hash value for Term within the range 1..Range. The allowed range is 1..2^27-1.

> **Warning:**
> This BIF is deprecated as the hash value may differ on different architectures. Also the hash values for integer terms larger than 2^27 as well as large binaries are very poor. The BIF is retained for backward compatibility reasons (it may have been used to hash records into a file), but all new code should use one of the BIFs erlang:phash/2 or erlang:phash2/1,2 instead.

hd(List)

Returns the first element of List.

```
> hd([1,2,3,4,5]).
1
```

Allowed in guard tests.

Failure: badarg if List is the empty list [], or is not a list.

erlang:hibernate(Module, Function, ArgumentList)

erlang:hibernate/3 gives a way to put a process into a wait state where its memory allocation has been reduced as much as possible, which is useful if the process does not expect to receive any messages in the near future.

The process will be awaken when a message is sent to it, and control will resume in Module:Function with the arguments given by ArgumentList with the call stack emptied, meaning that the process will terminate when that function returns. Thus erlang:hibernate/3 will never return to its caller.

If the process has any message in its message queue, the process will be awaken immediately in the same way as described above.

In more technical terms, what erlang:hibernate/3 will do is the following. It will discard the call stack for the process. Then it will garbage collect the process. After the garbage collection, all live data will be in one contionous heap. The heap will then be shrunk to the exact same size as the live data which it holds (even if that size should be less than the minimum heap size for the process).

If the size of the live data in the process is less than the minimum heap size, the first garbage collection occurring after the process has been awaken will ensure that the heap size is changed to a size not smaller than the minimum heap size.

Failure: badarg if Module or Function is not an atom, or if ArgumentList is not a list.

erlang:info(What)

This BIF is now equivalent to erlang:system_info/1 [page 114].

integer_to_list(Integer)

Returns a list of integers (ASCII codes) which correspond to Integer.

```
> integer_to_list(77).
"77"
```

Failure: badarg if Integer is not an integer.

erlang:integer_to_list(Integer, Base)

Returns a list of integers (ASCII codes) in base Base which correspond to Integer.

```
> erlang:integer_to_list(1023, 16).
"3FF"
```

Failure: badarg if Integer or Base is not an integer, or if Base is not in the range 2..36.

is_alive()

Returns true if the current node is alive; i.e., if the node can be part of a distributed system. Otherwise, it returns false.

is_atom(Term) -> Bool

Types:

- Term = term()
- Bool = true | false

Returns true if Term is an atom; otherwise returns false. This BIF may be used also in guards.

is_binary(Term) -> Bool

Types:

- Term = term()
- Bool = true | false

Returns true if Term is a binary; otherwise returns false. This BIF may be used also in guards.

is_boolean(Term) -> Bool

Types:

- Term = term()
- Bool = true | false

Returns `true` if `Term` is either the atom `true` or the atom `false` (i.e. a boolean); otherwise returns `false`. This BIF may be used also in guards.

`erlang:is_builtin(Module, Function, Arity)`

Returns `true` if `Module:Function/Arity` is a BIF implemented in C; otherwise returns `false`. This BIF is useful for builders of cross reference tools.

`is_float(Term) -> Bool`

Types:

- Term = term()
- Bool = true | false

Returns `true` if `Term` is a floating point number; otherwise returns `false`. This BIF may be used also in guards.

`is_function(Term) -> Bool`

Types:

- Term = term()
- Bool = true | false

Returns `true` if `Term` is a fun; otherwise returns `false`. This BIF may be used also in guards.

`is_integer(Term) -> Bool`

Types:

- Term = term()
- Bool = true | false

Returns `true` if `Term` is an integer; otherwise returns `false`. This BIF may be used also in guards.

`is_list(Term) -> Bool`

Types:

- Term = term()
- Bool = true | false

Returns `true` if `Term` is a list with zero or more elements; otherwise returns `false`. This BIF may be used also in guards.

`is_number(Term) -> Bool`

Types:

- Term = term()
- Bool = true | false

Returns `true` if `Term` is either an integer or a floating point number; otherwise returns `false`. This BIF may be used also in guards.

`is_pid(Term) -> Bool`

Types:

- Term = term()
- Bool = true | false

Returns `true` if `Term` is a process identifier (pid); otherwise returns `false`. This BIF may be used also in guards.

`is_port(Term) -> Bool`

Types:

- Term = term()
- Bool = true | false

Returns `true` if `Term` is a port; otherwise returns `false`. This BIF may be used also in guards.

`is_process_alive(Pid)`

`Pid` must refer to a process at the current node. Returns `true` if the process is alive, i.e., has not exited. Otherwise, returns `false`.

`is_record(Term, RecordTag) -> Bool`

Types:

- Term = term()
- RecordTag = atom()
- Bool = true | false

`RecordTag` must be an atom. Returns `true` if `Term` is a tuple and its first element is `RecordTag`. Otherwise, returns `false`.

> **Note:**
> Normally the compiler treats calls to `is_record/2` specially. It emits code to verify that `Term` is a tuple, that its first element is `RecordTag`, and that the size is correct. However, if the `RecordTag` is not a literal atom, the `is_record/2` BIF will be called instead.

If `RecordTag` is a literal atom, this BIF can also be used in a guard.

Failure: `badarg` if `RecordTag` is not an atom.

`erlang:is_record(Term, RecordTag, Size) -> Bool`

Types:

- Term = term()
- RecordTag = atom()
- Size = integer()
- Bool = true | false

The RecordTag must be an atom. Returns true if Term is a tuple, its first element is RecordTag, and its size is Size. Otherwise, returns false.

> **Note:**
> This BIF is documented for completeness. In most cases you should use is_record/2.

Failure: badarg if RecordTag is not an atom, or Size is not an integer.

`is_reference(Term) -> Bool`

Types:

- Term = term()
- Bool = true | false

Returns true if Term is a reference; otherwise returns false. This BIF may be used also in guards.

`is_tuple(Term) -> Bool`

Types:

- Term = term()
- Bool = true | false

Returns true if Term is a tuple; otherwise returns false. This BIF may be used also in guards.

`length(List)`

Returns the length of List.

```
> length([1,2,3,4,5,6,7,8,9]).
9
```

Allowed in guard tests.

Failure: badarg if List is not a proper list.

`link(Pid)`

Creates a link to the process (or port) Pid, if there is not such a link already. If a process attempts to create a link to itself, nothing is done. Returns true.

Sends an EXIT signal with reason noproc to the calling process if Pid does no longer exist.

Failure: badarg if Pid is not a pid or port.

`list_to_atom(List)`

Returns an atom whose text representation is the integers (Latin-1 codes) in List.

```
> list_to_atom([69, 114, 108, 97, 110, 103]).
'Erlang'
```

Failure: badarg if the argument is not a list of integers in the range [0, 255].

`list_to_binary(DeepList)`

Types:

- DeepList = iolist()

Returns a binary which is made from the integers and binaries in List. List may be deep and may contain any combination of integers and binaries.

Example: `list_to_binary([Bin1,1,[2,3,Bin2],4|Bin3])`

Failure: badarg if List is not a list, or if it or any sublist contains anything else than binaries or integers in the range [0, 255].

`list_to_float(List)`

Returns a float whose text representation is the integers (ASCII-values) in List.

```
> list_to_float([50,46,50,48,49,55,55,54,52,101,43,48]).
2.20178
```

Failure: badarg if List is not a list of integers, or if it contains a bad representation of a float.

`list_to_integer(List)`

Returns an integer whose text representation is the integers (ASCII-values) in List.

```
> list_to_integer([49, 50, 51]).
123
```

Failure: badarg if List is not a list of integers, or if it contains a bad representation of an integer.

`erlang:list_to_integer(List, Base)`

Returns an integer whose text representation in base Base is the integers (ASCII-values) in List.

```
> erlang:list_to_integer("3FF", 16).
1023
```

Failure: badarg if List is not a list of integers, contains a bad representation of an integer, or if Base is not in the range 2..36.

`list_to_pid(List)`

Returns a pid whose text representation is the integers (ASCII-values) in List.

> **Warning:**
> This BIF is intended for debugging and for use in the Erlang operating system. It should not be used in application programs.

```
> list_to_pid("<0.4.1>").
<0.4.1>
```

Failure: badarg if List is not a list of integers, or if it contains a bad representation of a pid.

list_to_tuple(List)

Returns a tuple which corresponds to List. List can contain any Erlang terms.

```
> list_to_tuple([mary, had, a, little, {dog, cat, lamb}]).
{mary, had, a, little, {dog, cat, lamb}}
```

Failure: badarg if List is not a proper list.

load_module(Module, Binary)

If Binary contains the object code for the module Module, this BIF loads that object code. Also, if the code for the module Module already exists, all export references are replaced so they point to the newly loaded code. The previously loaded code is kept in the system as old code, as there may still be processes which are executing that code. It returns either {module,Module}, where Module is the name of the module which has been loaded, or {error, Reason} if loading fails. Reason is one of the following:

badfile The object code in Binary has an incorrect format.

not_purged Binary contains a module which cannot be loaded because old code for this module already exists (see the BIFs purge_module and delete_module).

badfile The object code contains code for another module than Module

Failure: badarg if Module is not an atom, or Binary is not a binary.

> **Warning:**
> This BIF is intended for code server (see code(3)) and should not be used elsewhere.

erlang:loaded()

Returns a list of all loaded Erlang modules, including preloaded modules. A module will be included in the list if it has either current code or old code or both loaded.

erlang:localtime()

Returns the current local date and time {{Year, Month, Day}, {Hour, Minute, Second}}.

The time zone and daylight saving time correction depend on the underlying OS.

```
> erlang:localtime().
{{1996,11,6},{14,45,17}}
```

erlang:localtime_to_universaltime(DateTime)

Converts local date and time in DateTime to Universal Time Coordinated (UTC), if this is supported by the underlying OS. Otherwise, no conversion is done and DateTime is returned. The return value is of the form {{Year, Month, Day}, {Hour, Minute, Second}}.

Failure: badarg if DateTime is not a valid date and time tuple {{Year,Month,Day}, {Hour,Minute,Second}}.

```
> erlang:localtime_to_universaltime({{1996,11,6},{14,45,17}}).
{{1996,11,6},{13,45,17}}
```

erlang:localtime_to_universaltime(DateTime, IsDst)

Converts local date and time in DateTime to Universal Time Coordinated (UTC) just like erlang:localtime_to_universaltime/1, but the caller decides if daylight saving time is active or not.

If IsDst == true the DateTime is during daylight saving time, if IsDst == false it is not, and if IsDst == undefined the underlying OS may guess, which is the same as calling erlang:localtime_to_universaltime(DateTime).

Failure: badarg if DateTime is not a valid date and time tuple {{Year,Month,Day}, {Hour,Minute,Second}} or if IsDst is not one of the atoms true, false or undefined.

```
> erlang:localtime_to_universaltime({{1996,11,6},{14,45,17}}, true).
{{1996,11,6},{12,45,17}}
> erlang:localtime_to_universaltime({{1996,11,6},{14,45,17}}, false).
{{1996,11,6},{13,45,17}}
> erlang:localtime_to_universaltime({{1996,11,6},{14,45,17}}, undefined).
{{1996,11,6},{13,45,17}}
```

make_ref()

Returns an almost unique reference.

The returned reference will reoccur after approximately 2^82 calls; therefore it is unique enough for practical purposes.

```
> make_ref().
#Ref<0.0.0.135>
```

erlang:make_tuple(Arity, InitialValue)

Returns a new tuple of the given Arity, where all elements are InitialValue.

```
> erlang:make_tuple(4, []).
{[],[],[],[]}
```

erlang:md5(Data) -> Digest

Types:

- Data = iolist() | binary()
- Digest = binary()

Computes an MD5 message digest from Data, where the length of the digest is 128 bits (16 bytes). Data is a binary or a list of small integers and binaries.

See The MD5 Message Digest Algorithm (RFC 1321) for more information about MD5.

Failure: badarg if Data is not a list, or if it or any sublist contains anything else than binaries or integers in the range [0, 255].

`erlang:md5_final(Context) -> Digest`

Types:

- Context = Digest = binary()

Finishes the update of an MD5 Context and returns the computed MD5 message digest.

`erlang:md5_init() -> Context`

Types:

- Context = binary()

Creates an MD5 context, to be used in subsequent calls to md5_update/2.

`erlang:md5_update(Context, Data) -> NewContext`

Types:

- Data = iolist() | binary()
- Context = NewContext = binary()

Updates an MD5 Context with Data, and returns a NewContext.

`erlang:memory() -> MemList`

Types:

- MemList = [MemInfo]
- MemInfo = {atom(), int()}

Returns information about memory dynamically allocated by the Erlang emulator.

A list of tuples is returned. Each tuple has two elements. The first element is an atom describing memory type. The second element is memory size in bytes. A description of each tuple follows:

`total` The total amount of memory currently allocated. total is the sum of processes and system.

`processes` The total amount of memory currently allocated by the Erlang processes.

`processes_used` The total amount of memory currently used by the Erlang processes. This memory is part of the memory presented as processes memory.

`system` The total amount of memory currently allocated by the emulator that is not directly related to any Erlang process.

Memory presented as processes is not included in this memory.

`atom` The total amount of memory currently allocated for atoms.

This memory is part of the memory presented as system memory.

`atom_used` The total amount of memory currently used for atoms.

This memory is part of the memory presented as atom memory.

`binary` The total amount of memory currently allocated for binaries.

This memory is part of the memory presented as `system` memory.

`code` The total amount of memory currently allocated for Erlang code.

This memory is part of the memory presented as `system` memory.

`ets` The total amount of memory currently allocated for ets tables.

This memory is part of the memory presented as `system` memory.

`maximum` The maximum total amount of memory allocated since the emulator was started.

This tuple is only present when the emulator is run with instrumentation.

For information on how to run the emulator with instrumentation see the [instrument(3)] and/or [erl(1)] man pages.

> Note:
> The `system` value is not complete. Some allocated memory that should be part of the `system` value are not. For example, memory allocated by drivers is missing.
>
> When the emulator is run with instrumentation, the `system` value is more accurate, but memory directly allocated by `malloc` (and friends) are still not part of the `system` value. Direct calls to `malloc` are only done from OS specific runtime libraries and perhaps from user implemented Erlang drivers that do not use the memory allocation functions in the driver interface.
>
> Since the `total` value is the sum of `processes` and `system` the error in `system` will propagate to the `total` value.

The different values has the following relation to each other. Values beginning with an uppercase letter is not part of the result.

```
total = processes + system
processes = processes_used + ProcessesNotUsed
system = atom + binary + code + ets + OtherSystem
atom = atom_used + AtomNotUsed

RealTotal = processes + RealSystem
RealSystem = system + MissedSystem
```

> Note:
> The `total` value is supposed to be the total amount of memory dynamically allocated by the emulator. Shared libraries, the code of the emulator itself, and the emulator stack(s) are not supposed to be included. That is, the `total` value is *not* supposed to be equal to the total size of all pages mapped to the emulator. Furthermore, due to fragmentation and pre-reservation of memory areas, the size of the memory segments which contain the dynamically allocated memory blocks can be substantially larger than the total size of the dynamically allocated memory blocks.

More tuples in the returned list may be added in the future.

`erlang:memory(MemoryTypeSpec) -> MemList | int()`

Types:

- MemoryTypeSpec = MemoryType | [MemoryType]
- MemoryType = atom()
- MemList = [MemInfo]
- MemInfo = {atom(), int()}

`erlang:memory/1` returns the same type of information as `erlang:memory/0`, but allows the caller to select specific information.

`MemoryType` is an atom equal to any atom that is used by `erlang:memory/0` to describe a memory type.

When `MemoryTypeSpec` is an atom the corresponding memory size is returned as an integer.

When `MemoryTypeSpec` is a list of atoms the corresponding values are returned as a `MemList`. The elements of the list returned are sorted, with regard to the atoms, in the same order as the `MemoryTypeSpec` list is sorted with the exception that duplicate atoms are ignored.

Failure: `badarg` if `MemoryType` is not an atom that is used by `erlang:memory/0` to describe a memory type, or if the emulator is not run with instrumentation and `maximum` is used as a `MemoryType`.

`module_loaded(Module)`

Returns `true` if the module `Module` is loaded, otherwise returns `false`. It does not attempt to load the module.

> **Warning:**
> This BIF is intended for code server (see `code(3)`) and should not be used elsewhere. Use `code:is_loaded/1` instead.

```
> erlang:module_loaded(lists).
true
```

Failure: `badarg` if `Module` is not an atom.

`erlang:monitor(Type, Item) -> MonitorReference`

Types:

- Type = atom()
- Item = pid() | {RegisteredName,NodeName} | RegisteredName
- RegisteredName = atom()
- NodeName = atom()
- MonitorReference = reference()

The current process starts monitoring `Item` which is an object of type `Type`.

Currently only processes can be monitored, i.e. the only allowed `Type` is `process`, but other types may be allowed in the future.

Valid `Items` when `Type` is `process` are:

pid() The pid of the process to monitor.

{RegisteredName, NodeName} A tuple consisting of a registered name of a process and a node name. The process residing on the node NodeName with the registered name RegisteredName will be monitored.

RegisteredName The same as {RegisteredName, node()}.

> Note:
> When a process is monitored by registered name, the process that has the registered name at the time when erlang:monitor/2 is called will be monitored. The monitor will not be effected, if the registered name is unregistered.

A 'DOWN' message will be sent to the monitoring process if Item dies, if Item does not exist, or if the connection is lost to the node which Item resides on. A 'DOWN' message has the following pattern:

{'DOWN', MonitorReference, Type, Object, Info}

where:

MonitorReference The reference returned by erlang:monitor/2.

Type The type of the monitored object.

Object A reference to the monitored object.

When Type is process, Object will be:

- the pid of the monitored process, if Item was the pid in the call to erlang:monitor/2.
- {RegisteredName, NodeName}, if Item was {RegisteredName, NodeName} in the call to erlang:monitor/2.
- {RegisteredName, NodeName}, if Item was RegisteredName in the call to erlang:monitor/2. NodeName will in this case be the name of the local node.

Info When Type is process, Info is either the exit reason of the process, noproc (non-existing process), or noconnection (no connection to Items node).

> Note:
> If/when erlang:monitor/2 is extended (e.g. to handle other item types than process), other possible values for Type, Object, and Info in the 'DOWN' message will be introduced.

The monitoring is turned off either when the 'DOWN' message is sent, or when erlang:demonitor(MonitorReference) [page 77] is called (MonitorReference is the value returned by erlang:monitor/2).

If an attempt is made to monitor a process on an older node (where remote process monitoring is not implemented or one where remote process monitoring by registered name is not implemented), the call fails with badarg.

Making several calls to erlang:monitor/2 for the same Item is not an error; it results in several completely independent monitorings.

> **Note:**
> The format of the 'DOWN' message changed in the 5.2 version of the emulator (OTP release R9B) for monitor *by registered name*. The Object element of the 'DOWN' message could in earlier versions sometimes be the pid of the monitored process and sometimes be the registered name. Now the Object element is always a tuple consisting of the registered name and the node name. Processes on new nodes (emulator version 5.2 or greater) will always get 'DOWN' messages on the new format even if they are monitoring processes on old nodes. Processes on old nodes will always get 'DOWN' messages on the old format.

`monitor_node(Node, Flag)`

Monitors the status of the node Node. If Flag is true, monitoring is turned on; if Flag is false, monitoring is turned off. Calls to the BIF are accumulated. This is shown in the following example, where a process is already monitoring the node Node and a library function is called:

```
monitor_node(Node, true),
    ...  some operations
monitor_node(Node, false),
```

After the call, the process is still monitoring the node.

If Node fails or does not exist, the message {nodedown, Node} is delivered to the process. If a process has made two calls to monitor_node(Node, true) and Node terminates, two nodedown messages are delivered to the process. If there is no connection to Node, there will be an attempt to create one. If this fails, a nodedown message is delivered.

Nodes connected through hidden connections can be monitored as any other node with erlang:monitor_node/2.

Returns true.

Failure: badarg if Flag is not true or false, or if Node is not an atom indicating a remote node, or if the local node is not alive.

`node()`

Returns the name of the current node. If it is not a networked node but a local Erlang runtime system, the atom nonode@nohost is returned.

Allowed in guard tests.

`node(Arg)`

Returns the node where Arg is located. Arg can be a pid, a reference, or a port.

Allowed in guard tests.

Failure: badarg if Arg is not a pid, reference, or port.

`nodes()`

Returns a list of all visible nodes in the system, excluding the current node. Same as `nodes(visible)`.

`nodes(Arg)`

Types:

- Arg = ArgList | ArgAtom
- ArgList = [ArgAtom]
- ArgAtom = visible | hidden | connected | this | known

Returns a list of nodes according to argument given. The result returned when `Arg` is an `ArgList` is the list of nodes satisfying the disjunction(s) of `ArgAtoms` in `ArgList`.

`ArgAtom` description:

`visible` Nodes connected to this node through normal connections.

`hidden` Nodes connected to this node through hidden connections.

`connected` Nodes connected to this node.

`this` This node.

`known` Nodes which are known to this node, i.e., connected, previously connected, etc.

More `ArgAtoms` may be added in the future.

Some equalities: `[node()] = nodes(this)`, `nodes(connected) = nodes([visible, hidden])`, and `nodes() = nodes(visible)`.

Failure: `badarg` if `Arg` is not a valid `ArgAtom`, or a list of valid`ArgAtoms`.

`now()`

Returns the tuple {`MegaSecs`,`Secs`,`Microsecs`} which is the elapsed time since 00:00 GMT, January 1, 1970 (zero hour) on the assumption that the underlying OS supports this. Otherwise, some other point in time is chosen. It is also guaranteed that subsequent calls to this BIF returns continuously increasing values. Hence, the return value from `now()` can be used to generate unique time-stamps. It can only be used to check the local time of day if the time-zone info of the underlying operating system is properly configured.

`open_port(PortName, PortSettings)`

Returns a port identifier as the result of opening a new Erlang port. A port can be seen as an external Erlang process. `PortName` is one of the following:

{`spawn, Command`} Starts an external program. `Command` is the name of the external program which will be run. `Command` runs outside the Erlang work space unless an Erlang driver with the name `Command` is found. If found, that driver will be started. A driver runs in the Erlang workspace, which means that it is linked with the Erlang runtime system.

When starting external programs on Solaris, the system call `vfork` is used in preference to `fork` for performance reasons, although it has a history of being less robust. If there are problems with using `vfork`, setting the environment variable `ERL_NO_VFORK` to any value will cause `fork` to be used instead.

`Atom` This use of `open_port()` is obsolete and will be removed in a future version of Erlang. Use the `file` module instead.

{fd, In, Out} Allows an Erlang process to access any currently opened file descriptors used by Erlang. The file descriptor In can be used for standard input, and the file descriptor Out for standard output. It is only used for various servers in the Erlang operating system (shell and user). Hence, its use is very limited.

PortSettings is a list of settings for the port. Valid values are:

{packet, N} Messages are preceded by their length, sent in N bytes, with the most significant byte first. Valid values for N are 1, 2, or 4.

stream Output messages are sent without packet lengths. A user-defined protocol must be used between the Erlang process and the external object.

{line, N} Messages are delivered on a per line basis. Each line (delimited by the OS-dependent newline sequence) is delivered in one single message. The message data format is {Flag, Line}, where Flag is either eol or noeol and Line is the actual data delivered (without the newline sequence).

N specifies the maximum line length in bytes. Lines longer than this will be delivered in more than one message, with the Flag set to noeol for all but the last message. If end of file is encountered anywhere else than immediately following a newline sequence, the last line will also be delivered with the Flag set to noeol. In all other cases, lines are delivered with Flag set to eol.

The {packet, N} and {line, N} settings are mutually exclusive.

{cd, Dir} This is only valid for {spawn, Command}. The external program starts using Dir as its working directory. Dir must be a string. Not available on VxWorks.

{env, Environment} This is only valid for {spawn, Command}. The environment of the started process is extended using the environment specifications in Environment. Environment should be a list of tuples {Name,Value}, where Name is the name of an environment variable, and Value is the value it is to have in the spawned port process. Both Name and Value must be strings. The one exception is Value being the atom false (in analogy with os:getenv/1), which removes the environment variable. Not available on VxWorks.

exit_status This is only valid for {spawn, Command} where Command refers to an external program.

When the external process connected to the port exits, a message of the form {Port,{exit_status,Status}} is sent to the connected process, where Status is the exit status of the external process. If the program aborts, on Unix the same convention is used as the shells do (i.e., 128+signal).

If the eof option has been given as well, the eof message and the exit_status message appear in an unspecified order.

If the port program closes its stdout without exiting, the exit_status option will not work.

use_stdio This is only valid for {spawn, Command}. It allows the standard input and output (file descriptors 0 and 1) of the spawned (UNIX) process for communication with Erlang.

nouse_stdio The opposite of the above. Uses file descriptors 3 and 4 for communication with Erlang.

stderr_to_stdout Affects ports to external programs. The executed program gets its standard error file redirected to its standard output file. stderr_to_stdout and nouse_stdio are mutually exclusive.

in The port can only be used for input.

out The port can only be used for output.

`binary` All I/O from the port are binary data objects as opposed to lists of bytes.

`eof` The port will not be closed at the end of the file and produce an EXIT signal. Instead, it will remain open and a {`Port, eof`} message will be sent to the process holding the port.

The default is `stream` for all types of port and `use_stdio` for spawned ports.

Failure: `badarg` if the format of `PortName` or `PortSettings` is incorrect. If the port cannot be opened, the exit reason is the Posix error code which most closely describes the error, or `einval` if no Posix code is appropriate. The following Posix error codes may appear:

`enomem` There was not enough memory to create the port.

`eagain` There are no more available operating system processes.

`enametoolong` The external command given was too long.

`emfile` There are no more available file descriptors.

`enfile` A file or port table is full.

During use of a port opened using {`spawn,Name`}, errors arising when sending messages to it are reported to the owning process using signals of the form {`'EXIT',Port,PosixCode`}. Posix codes listed in the documentation for the `file` module.

The maximum number of ports that can be open at the same time is 1024 by default, but can be configured by the environment variable `ERL_MAX_PORTS`.

`erlang:phash(Term, Range)`

Portable hash function that will give the same hash for the same Erlang term regardless of machine architecture and ERTS version (the BIF was introduced in ERTS 4.9.1.1). Range can be between 1 and 2^32, the function returns a hash value for `Term` within the range `1..Range`.

This BIF could be used instead of the old deprecated `erlang:hash/2` BIF, as it calculates better hashes for all datatypes, but consider using `phash2/1,2` instead.

`erlang:phash2(Term [, Range])`

Portable hash function that will give the same hash for the same Erlang term regardless of machine architecture and ERTS version (the BIF was introduced in ERTS 5.2). Range can be between 1 and 2^32, the function returns a hash value for `Term` within the range `0..Range-1`. When called without the `Range` argument, a value in the range `0..2^27-1` is returned.

This BIF should always be used for hashing terms. It distributes small integers better than `phash/2`, and it is faster for bignums and binaries.

Note that the range `0..Range-1` is different from the range of `phash/2` (`1..Range`).

`pid_to_list(Pid)`

Returns a list which corresponds to the process Pid.

> **Warning:**
> This BIF is intended for debugging and for use in the Erlang operating system. It should not be used in application programs.

```
> pid_to_list(whereis(init)).
"<0.0.0>"
```

Failure: badarg if Pid is not a pid.

`port_close(Port)`

Closes an open port. Roughly the same as `Port ! {self(), close}` except for the error behaviour (see below), and that the port does *not* reply with `{Port, closed}`. Any process may close a port with `port_close/1`, not only the port owner (the connected process).

Returns: `true`.

Failure: `badarg` if `Port` is not an open port or the registered name of an open port.

For comparison: `Port ! {self(), close}` fails with `badarg` if `Port` cannot be sent to (i.e., `Port` refers neither to a port nor to a process). If `Port` is a closed port nothing happens. If `Port` is an open port and the current process is the port owner, the port replies with `{Port, closed}` when all buffers have been flushed and the port really closes, but if the current process is not the port owner the *port owner* fails with `badsig`.

Note that any process can close a port using `Port ! {PortOwner, close}` just as if it itself was the port owner, but the reply always goes to the port owner.

In short: `port_close(Port)` has a cleaner and more logical behaviour than `Port ! {self(), close}`.

`port_command(Port, Data)`

Types:

- Port = port()
- Data = iolist() | binary()

Sends data to a port. Same as `Port ! {self(), {command, Data}}` except for the error behaviour (see below). Any process may send data to a port with `port_command/2`, not only the port owner (the connected process).

Returns: `true`.

Failure: `badarg` if `Port` is not an open port or the registered name of an open port, or if `Data` is not an I/O list. An I/O list is binary or a (possibly) deep list of binaries or integers in the range 0 through 255.

For comparison: `Port ! {self(), {command, Data}}` fails with `badarg` if `Port` cannot be sent to (i.e., `Port` refers neither to a port nor to a process). If `Port` is a closed port the data message disappears without a sound. If `Port` is open and the current process is not the port owner, the *port owner* fails with `badsig`. The port owner fails with`badsig` also if `Data` is not a legal I/O list.

Note that any process can send to a port using `Port ! {PortOwner, {command, Data}}` just as if it itself was the port owner.

In short: `port_command(Port, Data)` has a cleaner and more logical behaviour than `Port ! {self(), {command, Data}}`.

`port_connect(Port, Pid)`

Sets the port owner (the connected port) to `Pid`. Roughly the same as `Port ! {self(), {connect, Pid}}` except for the following:

- The error behavior differs, see below.
- The port does *not* reply with `{Port,connected}`.
- The new port owner gets linked to the port.

The old port owner stays linked to the port and have to call `unlink(Port)` if this is not desired. Any process may set the port owner to be any process with `port_connect/2`.

Returns: `true`.

Failure: `badarg` if `Port` is not an open port or the registered name of a port, or if `Pid` is not a valid local pid.

For comparison: `Port ! {self(), {connect, Pid}}` fails with `badarg` if `Port` cannot be sent to (i.e., `Port` refers neither to a port nor to a process). If `Port` is a closed port nothing happens. If `Port` is an open port and the current process is the port owner, the port replies with `{Port, connected}` to the old port owner. Note that the old port owner is still linked to the port, and that the new is not. If `Port` is an open port and the current process is not the port owner, the *port owner* fails with `badsig`. The port owner fails with `badsig` also if `Pid` is not a valid local pid.

Note that any process can set the port owner using `Port ! {PortOwner, {connect, Pid}}` just as if it itself was the port owner, but the reply always goes to the port owner.

In short: `port_connect(Port, Pid)` has a cleaner and more logical behaviour than `Port ! {self(),{connect,Pid}}`.

`port_control(Port, Operation, Data)`

Types:

- Port = port()
- Operation = integer()
- Data = iolist() | binary()

Performs a synchronous control operation on a port. The meaning of `Operation` and `Data` depends on the port, i.e., on the port driver. Not all port drivers support this control feature.

Returns: a list of integers in the range 0 through 255, or a binary, depending on the port driver. The meaning of the returned data also depends on the port driver.

Failure: `badarg` if `Port` is not an open port or the registered name of a port, if `Operation` cannot fit in a 32-bit integer, if the port driver does not support synchronous control operations, if `Data` is not a valid I/O list (see port_command/2), or if the port driver so decides for any reason (probably something wrong with `Operation` or `Data`).

`erlang:port_call(Port, Operation, Data)`

Performs a synchronous call to a port. The meaning of Operation and Data depends on the port, i.e., on the port driver. Not all port drivers support this feature.

Port is an Erlang port, referring to a driver.

Operation is an integer, which is passed on to the driver.

Data is any Erlang term. This data is converted to binary term format and sent to the port.

Returns: a term from the driver. The meaning of the returned data also depends on the port driver.

Failure: badarg if Port is not an open port or the registered name of a port, if Operation cannot fit in a 32-bit integer, if the port driver does not support synchronous control operations, or if the port driver so decides for any reason (probably something wrong with Operation or Data).

erlang:port_info(Port, Item)

Returns information about the port Port as specified by Item, which can be any one of the atoms registered_name, id, connected, links, name, input, or output.

{registered_name,Atom} Atom is the registered name of the port. If the port has no registered name, this tuple is not present in the list.

{id,Index} Index is the internal index of the port. This index may be used to separate ports.

{connected, Pid} Pid is the process connected to the port.

{links,ListOfPids} ListOfPids is a list of Pids with processes to which the port has a link.

{name,String} String is the command name set by open_port.

{input,Bytes} Bytes is the total number of bytes read from the port.

{output,Bytes} Bytes is the total number of bytes written to the port.

All implementations may not support all of the above Items. Returns undefined if the port does not exist.

Failure: badarg if Port is not a process identifier, or if Port is a port identifier of a remote process.

erlang:port_to_list(Port)

Returns a list which corresponds to the port identifier Port.

> **Warning:**
> This BIF is intended for debugging and for use in the Erlang operating system. It should not be used in application programs.

```
> erlang:port_to_list(open_port({spawn,ls}, [])).
"#Port<0.15>"
```

Failure: badarg if Port is not a port identifier.

`erlang:ports()`

Returns a list of all ports on the current node.

`pre_loaded()`

Returns a list of Erlang modules which are pre-loaded in the system. As all loading of code is done through the file system, the file system must have been loaded previously. Hence, at least the module `init` must be pre-loaded.

`erlang:process_display(Pid, Type)`

Writes information about the local process `Pid` on standard error. The currently allowed value for the atom `Type` is `backtrace`, which shows the contents of the stack, including information about the call chain, with the most recent data printed last. The format of the output is not further defined.

`process_flag(Flag, Option)`

Sets certain flags for the process which calls this function. Returns the old value of the flag.

`process_flag(trap_exit, Boolean)` When `trap_exit` is set to `true`, EXIT signals arriving to a process are converted to {`'EXIT'`, `From`, `Reason`} messages, which can be received as ordinary messages. If `trap_exit` is set to `false`, the process exits if it receives an EXIT signal other than `normal` and the EXIT signal is propagated to its linked processes. Application processes should normally not trap exits.

`process_flag(error_handler, Module)` This is used by a process to redefine the error handler for undefined function calls and undefined registered processes. Inexperienced users should not use this flag since code autoloading is dependent on the correct operation of the error handling module.

`process_flag(min_heap_size, MinHeapSize)` This changes the minimum heap size for the current process.

`process_flag(priority, Level)` This sets the process priority. `Level` is an atom. All implementations support three priority levels, `low`, `normal`, and `high`. The default is `normal`.

`process_flag(save_calls, N)` `N` must be an integer in the interval [0, 10000]. If $N > 0$, call saving is made active for the process, which means that information about the `N` most recent global function calls, BIF calls, sends and receives made by the process are saved in a list, which can be retrieved with `process_info(Pid, last_calls)`. A global function call is one in which the module of the function is explicitly mentioned. Only a fixed amount of information is saved: a tuple {`Module`, `Function`, `Arity`} for function calls, and the mere atoms `send`, `'receive'` and `timeout` for sends and receives (`'receive'` when a message is received and `timeout` when a receive times out). If $N = 0$, call saving is disabled for the process, which is the default. Whenever the size of the call saving list is set, its contents are reset.

Failure: `badarg` if `Flag` is not an atom, or is not a recognized flag value, or if `Option` is not a recognized term for `Flag`.

`process_flag(Pid, Flag, Option)`

Sets certain flags for the process Pid, in the same manner as process_flag/2. Returns the old value of the flag. The allowed values for Flag are only a subset of those allowed in process_flag/2, namely: save_calls.

Failure: badarg if Pid is not a process on the local node, or if Flag is not an atom, or is not a recognized flag value, or if Option is not a recognized term for Flag.

process_info(Pid)

Returns a list containing tuples with information about the process Pid. The order of these tuples are not defined, nor are all the tuples mandatory.

> **Warning:**
> This BIF is intended for debugging only.

{current_function, {Module, Function, Args}} Module, Function, Args is the current function call of the process.

{dictionary, Dictionary} Dictionary is the dictionary of the process.

{error_handler, Module} Module is the error handler module used by the process (for undefined function calls, for example).

{group_leader, Groupleader} Groupleader is group leader for the I/O of the process.

{heap_size, Size} Size is the heap size of the process in heap words.

{initial_call, {Module, Function, Arity}} Module, Function, Arity is the initial function call with which the process was spawned.

{links, ListOfPids} ListOfPids is a list of Pids, with processes to which the process has a link.

{message_queue_len, MessageQueueLen} MessageQueueLen is the number of messages currently in the message queue of the process. This is the length of the list MessageQueue returned as the info item messages (see below).

{messages, MessageQueue} MessageQueue is a list of the messages to the process, which have not yet been processed.

{priority, Level} Level is the current priority level for the process. Only low and normal are always supported.

{reductions, Number} Number is the number of reductions executed by the process.

{registered_name, Atom} Atom is the registered name of the process. If the process has no registered name, this tuple is not present in the list.

{stack_size, Size} Size is the stack size of the process in stack words.

{status, Status} Status is the status of the process. Status is waiting (waiting for a message), running, runnable (ready to run, but another process is running), or suspended (suspended on a "busy" port or by the erlang:suspend_process/1 BIF).

{trap_exit, Boolean} Boolean is true if the process is trapping exits, otherwise it is false.

Failure: badarg if Pid is not a pid, or if it is the pid of a remote process.

process_info(Pid, Item)

Returns information about the process Pid as specified by Item, in the form {Item, Info}. Item can be any one of the atoms backtrace, current_function, dictionary, error_handler, group_leader, heap_size, initial_call, last_calls, links, memory, message_queue_len, messages, monitored_by, monitors, priority, reductions, registered_name, stack_size, status or trap_exit.

Returns undefined if no information is known about the process.

Item registered_name returns [] if the process has no registered name.

Item memory returns {memory,Size}, where Size is the size of the process in bytes. This includes stack, heap and internal structures.

Item backtrace returns a binary, which contains the same information as the output from erlang:process_display(Pid, backtrace). Use binary_to_list/1 to obtain the string of characters from the binary.

Item last_calls returns false if call saving is not active for the process (see process_flag/3 [page 103]). If call saving is active, a list is returned, in which the last element is the most recent.

Item links returns a list of pids to which the process is linked.

Item monitors returns a list of monitors (started by erlang:monitor/2) that are active for the process. For a local process monitor or a remote process monitor by pid, the list item is {process, Pid}, and for a remote process monitor by name, the list item is {process, {Name, Node}}.

Item monitored_by returns a list of pids that are monitoring the process (with erlang:monitor/2).

Not all implementations support every one of the above Items.

Failure: badarg if Pid is not a pid, or if it is a process identifier of a remote process.

processes()

Returns a list of all processes on the current node.

```
> processes().
[<0.0.0>,
 <0.2.0>,
 <0.4.0>,
 <0.5.0>,
 <0.7.0>,
 <0.8.0>]
```

purge_module(Module)

Removes old code for Module. Before this BIF is used, erlang:check_process_code/2 should be called to check that no processes are executing old code in this module.

> **Warning:**
> This BIF is intended for code server (see code(3)) and should not be used elsewhere.

Failure: badarg if Module does not exist.

put(Key, Value)

Adds a new Value to the process dictionary and associates it with Key. If a value is already associated with Key, that value is deleted and replaced by the new value Value. It returns any value previously associated with Key, or undefined if no value was associated with Key. Key and Value can be any valid Erlang terms.

> **Note:**
> The values stored when put is evaluated within the scope of a catch will not be retracted if a throw is evaluated, or if an error occurs.

```
> X = put(name, walrus), Y = put(name, carpenter),
  Z = get(name),
  {X, Y, Z}.
{undefined,walrus,carpenter}
```

erlang:raise(Class, Reason, Stacktrace) <func> <name>erlang:raise(Class, Reason, Stacktrace)

Stops the execution of the current process with an exception of given class, reason and stacktrace.

> **Warning:**
> This BIF is intended for debugging and for use in the Erlang operating system. It should not be used in application programs.

Class is one of error, exit or throw, so if it were not for the stacktrace erlang:raise(Class, Reason, Stacktrace) is equivalent to erlang:Class(Reason). Reason is any term and Stacktrace is a list as returned from get_stacktrace(), that is a list of 3-tuples {Module,Function,A} where Module and Function are atoms and A should be an integer arity or an argument list. The stacktrace may also contain {Fun,A} tuples where Fun is a fun and A is an argument list.

The stacktrace is used as the exception stacktrace for the current process, so it will be truncated to the current maximum stacktrace depth. Any non-existent functions or funs it refers to may also be removed, even if they became non-existent after the stacktrace was stored for example due to code purge. This is because a true stacktrace

can only refer to existing code. (Except for the head element from a `function_clause` or `undef` error that refers to the function that did not exist)

Because evaluating this function causes the process to terminate, it has no return value - unless the arguments are invalid, in which case the function *returns the error reason*, that is `badarg`. If you want to be really sure not to return you can call `erlang:error(erlang:raise(Class, Reason, Stacktrace))` and hope to distinguish exceptions later.

`erlang:read_timer(Ref)`

`Ref` is a timer reference returned by `send_after/3` or `start_timer/3`. If the timer is active, the function returns the time in milliseconds left until the timer will expire, otherwise `false` (which may mean that `Ref` was never a timer, or that it has been cancelled, or that it has already delivered its message).

Failure: `badarg` if `Ref` is not a reference.

`erlang:ref_to_list(Ref)`

Returns a list which corresponds to the reference `Ref`.

> **Warning:**
> This BIF is intended for debugging and for use in the Erlang operating system. It should not be used in application programs.

```
> erlang:ref_to_list(make_ref()).
"#Ref<0.0.0.134>"
```

Failure: `badarg` if `Ref` is not a reference.

`register(Name, P)`

Associates the name `Name` with the port or pid `P`. `Name`, which must be an atom, can be used instead of a port or pid in the send operator (`Name ! Message`).

Returns `true`.

Failure: `badarg` if `P` is not an active port or process, if `P` is on another node, if `Name` is already in use, if the port or process is already registered (already has a name), if `Name` is not an atom, or if `Name` is the atom `undefined`.

`registered()`

Returns a list of names which have been registered using `register/2`.

```
> registered().
[code_server, file_server, init, user, my_db]
```

`erlang:resume_process(Pid)`

Resume a suspended process.

> **Warning:**
> This BIF is intended for debugging only.

`round(Number)`

Returns an integer by rounding the number `Number`.

```
> round(5.5).
6
```

Allowed in guard tests.

Failure: `badarg` if `Number` is not a number.

`self()`

Returns the pid (process identity) of the calling process.

```
> self().
<0.26.0>
```

Allowed in guard tests.

`erlang:send(Dest, Msg)`

Sends a message and returns `Msg`. This is the same as `Dest ! Msg`.

`Dest` may be a remote or local pid, a (local) port, a locally registered name, or a tuple {`Name,Node`} for a registered name on another node.

`erlang:send(Dest, Msg, Options)`

Sends a message and returns `ok`, or does not send the message but returns something else (see below). Otherwise the same as `send/2`. See also `send_nosuspend/2,3` for more detailed explanation and warnings.

`Options` is a list of options. The possible options are:

`nosuspend` If the sender would have to be suspended to do the send, `nosuspend` is returned instead.

`noconnect` If the destination node would have to be autoconnected before doing the send, `noconnect` is returned instead.

As with `send_nosuspend/2,3`: *Use with extreme care!*

`erlang:send_after(Time, Pid, Msg)`

Time is a non-negative integer, Pid is either a pid or an atom, and Msg is any Erlang term. The function returns a reference Ref.

After Time ms, send_after/3 sends Msg to Pid.

If Pid is an atom, it is supposed to be the name of a registered process. The process referred to by the name is looked up at the time of delivery. No error is given if the name does not refer to a process. See also start_timer/3 and cancel_timer/1.

Limitations: Pid must be a process on the local node. The timeout value must fit in 32 bits.

Failure: badarg if any arguments are of the wrong type, or do not obey the limitations noted above.

erlang:send_nosuspend(Dest, Msg)

The same as send(Dest, Msg, [nosuspend), but returns true if the message was sent and false if the message was not sent because the sender would have been suspended.

This function is intended for send operations towards an unreliable remote node without ever blocking the sending (Erlang) process. If the connection to the remote node (usually not a real Erlang node, but a node written in C or Java) is overloaded, this function *will not send the message* but return false instead.

The same happens, if Dest refers to a local port that is busy. For all other destinations (allowed for the ordinary send operator '!') this function sends the message and returns true.

This function is only to be used in very rare circumstances where a process communicates with Erlang nodes that can disappear without any trace causing the TCP buffers and the drivers que to be overfull before the node will actually be shut down (due to tick timeouts) by net_kernel. The normal reaction to take when this happens is some kind of premature shutdown of the other node.

Note that ignoring the return value from this function would result in *unreliable* message passing, which is contradictory to the Erlang programming model. The message is *not* sent if this function returns false.

Note also that in many systems, transient states of overloaded queues are normal. The fact that this function returns false does not in any way mean that the other node is guaranteed to be nonrespoinsive, it could be a temporary overload. Also a return value of true does only mean that the message could be sent on the (TCP) channel without blocking, the message is not guaranteed to have arrived at the remote node. Also in the case of a disconnected nonresponsive node, the return value is true (mimics the behaviour of the ! operator). The expected behaviour as well as the actions to take when the function returns false are application and hardware specific.

Use with extreme care!

erlang:send_nosuspend(Dest, Msg, Options)

The same as `send(Dest, Msg, [nosuspend | Options])`, but with boolean return value.

This function behaves like `send_nosuspend/2`, but takes a third parameter, a `list()` of options. The only currently implemented option is `noconnect`. The option `noconnect` makes the function return `false` if the remote node is not currently reachable by the local Erlang node. The normal behaviour is to try to connect to the node, which may stall the process for a shorter period. The use of the `noconnect` option makes it possible to be absolutely sure not to get even the slightest delay when sending to a remote process. This is especially useful when communicating with nodes who expect to always be the connecting part (i.e. nodes written in C or Java).

Whenever the function returns `false` (either when a suspend would occur or when `noconnect` was specified and the node was not already connected), the message is guaranteed *not* to have been sent.

Use with extreme care!

`erlang:set_cookie(Node, Cookie)`

Sets the magic cookie of `Node` to the atom `Cookie`. If `Node` is the current node, the function also sets the cookie of all other unknown nodes to `Cookie` (see auth(3)).

`setelement(Index, Tuple, Value)`

Returns a tuple which is a copy of the argument `Tuple` with the element given by the integer argument `Index` (the first element is the element with index 1) replaced by the argument `Value`.

```
> setelement(2, {10, green, bottles}, red).
{10, red, bottles}
```

Failure: `badarg` if `Index` is not an integer, `Tuple` is not a tuple, or if `Index` is less than 1 or greater than the size of `Tuple`.

`size(Item)`

Returns an integer which is the size of the argument `Item`, which must be either a tuple or a binary.

```
> size({morni, mulle, bwange}).
3
```

Allowed in guard tests.

Failure: `badarg` if `Item` is not a tuple or a binary.

`spawn(Fun)`

Returns the pid of a new process started by the application of `Fun` to the empty argument list `[]`. Otherwise works like `spawn/3`.

`spawn(Node, Fun)`

Returns the pid of a new process started by the application of `Fun` to the empty argument list `[]` on node `Node`. Otherwise works like `spawn/4`.

spawn(Module, Function, ArgumentList)

Returns the pid of a new process started by the application of Module:Function to ArgumentList. *Note:* The new process created will be placed in the system scheduler queue and will be run some time later.

error_handler:undefined_function(Module, Function, ArgumentList) is evaluated by the new process if Module:Function/Arity does not exist (where Arity is the length of ArgumentList). The error handler can be redefined (see BIF process_flag/2)). Arity is the length of ArgumentList. If error_handler is undefined, or the user has redefined the default error_handler its replacement is undefined, a failure with the reason undef will occur.

```
> spawn(speed, regulator, [high_speed, thin_cut]).
<0.13.1>
```

Failure: badarg if Module and/or Function is not an atom, or if ArgumentList is not a list.

spawn(Node, Module, Function, ArgumentList)

Works like spawn/3, with the exception that the process is spawned at Node. If Node does not exist, a useless pid is returned.

Failure: badarg if Node, Module, or Function are not atoms, or ArgumentList is not a list.

spawn_link(Fun)

Works like spawn/1 except that a link is made from thed current process to the newly created one, atomically.

spawn_link(Node, Fun)

Works like spawn/2 except that a link is made from thed current process to the newly created one, atomically.

Returns the Pid of the newly created process.

Failure: See spawn/3.

spawn_link(Module, Function, ArgumentList)

This BIF is identical to the following code being evaluated in an atomic operation:

```
> Pid = spawn(Module, Function, ArgumentList),
  link(Pid),
  Pid.
```

This BIF is necessary since the process created might run immediately and fail before link/1 is called.

Returns the pid of the newly created process.

Failure: See spawn/3.

spawn_link(Node, Module, Function, ArgumentList)

Works like spawn_link/3, except that the process is spawned at Node. If an attempt is made to spawn a process on a node which does not exist, a useless Pid is returned, and an EXIT signal will be received.

spawn_opt(Fun, Options)

Returns the pid of a new process started by the application of Fun to the empty argument list []. Otherwise works like spawn_opt/4.

spawn_opt(Node, Fun, Options)

Returns the pid of a new process started by the application of Fun to the empty argument list []. Otherwise works like spawn_opt/5.

spawn_opt(Module, Function, ArgumentList, Options)

Works exactly like spawn/3, except that an extra option list can be given when creating the process.

> **Warning:**
> This BIF is only useful for performance tuning. Random tweaking of the parameters without measuring execution times and memory consumption may actually make things worse. Furthermore, most of the options are inherently implementation-dependent, and they can be changed or removed in future versions of OTP.

link Sets a link to the parent process (like spawn_link/3 does).

{priority, Level} Sets the priority of the new process. Equivalent to executing process_flag(priority, Level) in the start function of the new process, except that the priority will be set before the process is scheduled in the first time.

{fullsweep_after, Number} The Erlang runtime system uses a generational garbage collection scheme, using an "old heap" for data that has survived at least one garbage collection. When there is no more room on the old heap, a fullsweep garbage collection will be done.

Using the fullsweep_after option, you can specify the maximum number of generational collections before forcing a fullsweep even if there is still room on the old heap. Setting the number to zero effectively disables the general collection algorithm, meaning that all live data is copied at every garbage collection.

Here are a few cases when it could be useful to change fullsweep_after. Firstly, if you want binaries that are no longer used to be thrown away as soon as possible. (Set Number to zero.) Secondly, a process that mostly have short-lived data will be fullsweeped seldom or never, meaning that the old heap will contain mostly garbage. To ensure a fullsweep once in a while, set Number to a suitable value such as 10 or 20. Thirdly, in embedded systems with limited amount of RAM and no virtual memory, you might want to preserve memory by setting Number to zero. (You probably want to the set the value globally. See system_flag/2 [page 113].)

{min_heap_size, Size} Gives a minimum heap size in words. Setting this value higher than the system default might speed up some processes because less garbage collection is done. Setting too high value, however, might waste memory and slow down the system due to worse data locality. Therefore, it is recommended to use this option only for fine-tuning an application and to measure the execution time with various Size values.

spawn_opt(Node, Module, Function, ArgumentList, Options)

Works like spawn_opt/4, except that the process is spawned at Node. If an attempt is made to spawn a process on a node which does not exist, a useless pid is returned, and an EXIT signal will be received.

split_binary(Binary, Pos)

Returns a tuple which contains two binaries which are the result of splitting Binary into two parts at position Pos. This is not a destructive operation. After this operation, there are three binaries altogether. Returns a tuple consisting of the two new binaries. For example:

```
1> B = list_to_binary("0123456789").
<<48,49,50,51,52,53,54,55,56,57>>
2> size(B).
10
3> {B1, B2} = split_binary(B,3).
{<<48,49,50>>,
 <<51,52,53,54,55,56,57>>}
4> size(B1).
3
5> size(B2).
7
```

Failure: badarg if Binary is not a binary, or Pos is not an integer or is out of range.

erlang:start_timer(Time, Proc, Msg)

Time is a non-negative integer, Proc is either a pid or an atom, and Msg is any Erlang term. The function returns a reference.

After Time ms, start_timer/3 sends the tuple {timeout, Ref, Msg} to Proc, where Ref is the reference returned by start_timer/3.

If Proc is an atom, it is supposed to be the name of a registered process. The process referred to by the name is looked up at the time of delivery. No error is given if the name does not refer to a process. See also send_after/3 and cancel_timer/1.

Limitations: Proc must be a process on the local node. The timeout value must fit in 32 bits.

Failure: badarg if any arguments are of the wrong type, or do not obey the limitations noted above.

statistics(Type)

Returns information about the system. Type is an atom which is one of:

run_queue Returns the length of the run queue, that is the number of processes that are ready to run.

runtime Returns {Total_Run_Time,Time_Since_Last_Call}.

wall_clock Returns {Total_Wallclock_Time, Wallclock_Time_Since_Last_Call}. wall_clock can be used in the same manner as the atom runtime, except that real time is measured as opposed to runtime or CPU time.

reductions Returns {Total_Reductions, Reductions_Since_Last_Call}.

garbage_collection Returns {Number_of_GCs,Words_Reclaimed,0}. This information may not be valid for all implementations.

All times are in milliseconds.

```
> statistics(runtime).
{1690, 1620}
> statistics(reductions).
{2046, 11}
> statistics(garbage_collection).
{85, 23961, 0}
```

Failure: badarg if Type is not one of the atoms shown above.

erlang:suspend_process(Pid)

Suspends the process Pid.

> **Warning:**
> This BIF is intended for debugging only.

erlang:system_flag(Flag, Value)

This BIF sets various system properties of the Erlang node. If Flag is a valid name of a system flag, its value is set to Value, and the old value is returned.

The following values for Flag are currently allowed:

backtrace_depth Sets the maximum depth of call stack backtraces in the exit reason element of 'EXIT' tuples.

fullsweep_after The value of the fullsweep_after is an non-negative integer which indicates how many times generational garbages collections can be done without forcing a fullsweep collection. The value applies to new processes; processes already running are not affected.

In low-memory systems (especially without virtual memory), setting the value to zero can help to conserve memory.

An alternative way to set this value is through the (operating system) environment variable ERL_FULLSWEEP_AFTER.

min_heap_size Sets the default minimum heap size for processes. The size is given in words. The new min_heap_size only effects processes spawned after the change of min_heap_size has been made. The min_heap_size can be set for individual processes by use of spawn_opt() [page 111] or process_flag/2 [page 102].

`trace_control_word` Sets the value of the node's trace control word to `Value`. `Value` should be an unsigned integer. For more information see documentation of the [set_tcw] function in the [match specification] documentation in the ERTS User's Guide.

> **Note:**
> `erlang:system_flag/2` accepts other arguments than those documented above. These arguments have *intentionally* been left undocumented. This either because the undocumented `Flag` argument is used for implementation of other documented functionality, or is used for testing and debugging of the emulator. *Do not use the undocumented arguments* of `erlang:system_flag/2`, they may have undesirable side-effects, and may be changed or removed at any time without prior notice.

`erlang:system_info(What)`

Types:

- What = atom() | {atom() | term()}

This BIF returns various information about the current system (emulator).

`What` can be one of the following terms:

`allocated_areas` Information about miscellaneous allocated memory areas.

A list of tuples is returned. Each tuple contains an atom describing type of memory as first element and amount of allocated memory in bytes as second element. In those cases when there is information present about allocated and used memory, a third element is present. This third element contains the amount of used memory in bytes.

`erlang:system_info(allocated_areas)` is intended for debugging, and the content is highly implementation dependent. The content of the results will therefore change when needed without prior notice.

Note: The sum of these values is *not* the total amount of memory allocated by the emulator. Some values are part of other values, and some memory areas are not part of the result. If you are interested in the total amount of memory allocated by the emulator see erlang:memory/0 [page 91] or erlang:memory/1 [page 93].

`allocator` Returns {`Allocator, Version, Features, Settings`}.

Types:

- `Allocator = atom()`
- `Version = [int()]`
- `Features = [atom()]`
- `Settings = [{Subsystem, [{Parameter, Value}]}]`
- `Subsystem = atom()`
- `Parameter = atom()`
- `Value = term()`

Explanation:

- `Allocator` corresponds to the `malloc()` implementation used. If `Allocator` equals `undefined`, the `malloc()` implementation used could not be identified. Currently `elib_malloc` and `glibc` can be identified.

- `Version` is a list of integers (but not a string) representing the version of the `malloc()` implementation used.
- `Features` is a list of atoms representing allocation features used.
- `Settings` is a list of subsystems, their configurable parameters, and used values. Settings may differ between different combinations of platforms, allocators, and allocation features. Memory sizes are given in bytes.

See also the [”System Flags Effecting erts_alloc”] section in the [erts_alloc(3)] documentation.

`{allocator, Alloc}` Types:

- `Alloc = atom()`

Returns information about the `Alloc` allocator. If `Alloc` is not a recognized allocator, `undefined` is returned. If `Alloc` is disabled, `false` is returned.

Note: The information returned is highly implementation dependent and may be changed, or removed at any time without prior notice. It was initially intended as a tool when developing new allocators, but since it might be of interest for others it has been briefly documented.

The returned information more or less speaks for itself once you have read the [erts_alloc(3)] documentation, but it can be worth explaining some things. Call counts are presented by two values. The first value is giga calls, and the second value is calls. `mbcs`, and `sbcs` are abbreviations for, respectively, multiblock carriers, and singleblock carriers. Sizes are presented in bytes. When it is not a size that is presented, it is the amount of something. Sizes and amounts are often presented by three values, the first is current value, the second is maximum value since the last call to `erlang:system_info({allocator, Alloc})`, and the third is maximum value since the emulator was started. If only one value is present, it is the current value. `fix_alloc` memory block types are presented by two values. The first value is memory pool size and the second value used memory size.

`compat_rel` Returns the compatibility mode of the current node as an integer. The integer returned represents the Erlang/OTP release which the current emulator has been set to be backward compatible with. The compatibility mode can be configured at startup by use of the [+R] system flag (see the [erl(1)] documentation)

`creation` Returns the creation of the current node as an integer. The creation is changed when a node is restarted. The creation of a node is stored in process identifiers, port identifiers, and references. This makes it (to some extent) possible to distinguish between identifiers from different incarnations of a node. Currently valid creations are integers in the range [1, 3], but this may (probably will) change in the future. If the node is not alive 0 is returned.

`dist` Returns a binary containing a string of distribution information formatted as in Erlang crash dumps. For more information see the [”How to interpret the Erlang crash dumps”] chapter in the ERTS User's Guide.

`dist_ctrl` Returns a list of tuples `{NodeName,ControllingEntity}`, one entry for each connected remote node. The `NodeName` is the name of the node and the `ControllingEntity` is the `port()` or `pid()` responsible for the communication to that node. More specifically, the `ControllingEntity` for nodes connected via TCP/IP (the normal case) is the socket actually used in communication with the specific node.

`elib_malloc` If the emulator uses the `elib_malloc` memory allocator, a list of two-element tuples containing status information of `elib_malloc` is returned; otherwise, `false` is returned. The list currently contains the following two-element tuples (all sizes are presented in bytes):

{heap_size, Size} Where Size is the current heap size.

{max_alloced_size, Size} Where Size is the maximum amount of memory allocated on the heap since the emulator started.

{alloced_size, Size} Where Size is the current amount of memory allocated on the heap.

{free_size, Size} Where Size is the current amount of free memory on the heap.

{no_alloced_blocks, No} Where No is the current number of allocated blocks on the heap.

{no_free_blocks, No} Where No is the current number of free blocks on the heap.

{smallest_alloced_block, Size} Where Size is the size of the smallest allocated block on the heap.

{largest_free_block, Size} Where Size is the size of the largest free block on the heap.

fullsweep_after Returns {fullsweep_after, integer()} which is the fullsweep_after garbage collection setting used by default. For more information see the garbage_collection [page 116] argument described below.

garbage_collection Returns a list describing the default garbage collection settings. A process spawned on the current node by a spawn() or spawn_link() will use these garbage collection settings. The default settings can be changed by use of system_flag/2 [page 113]. spawn_opt() [page 111] can spawn a process that does not use the default settings.

global_heaps_size Returns the current size of the shared (global) heap.

heap_sizes Returns a list of integers representing valid heap sizes in words. All Erlang heaps are sized from sizes in this list.

heap_type Returns the heap type used by the current emulator. Currently the following heap types exist:

private Each process has a heap reserved for its use and no references between heaps of different processes are allowed. Messages passed between processes are copied between heaps.

shared One heap for use by all processes. Messages passed between processes are passed by reference.

hybrid A hybrid of the private and shared heap types. A shared heap as well as private heaps are used.

info Returns a binary containing a string of miscellaneous system information formatted as in Erlang crash dumps. For more information see the [”How to interpret the Erlang crash dumps”] chapter in the ERTS User's Guide.

loaded Returns a binary containing a string of loaded module information formatted as in Erlang crash dumps. For more information see the [”How to interpret the Erlang crash dumps”] chapter in the ERTS User's Guide.

machine Returns a string containing the Erlang machine name.

process_count Returns the number of processes currently existing on the current node as an integer. The same value as length(processes()) returns.

process_limit Returns the maximum number of concurrently existing processes on the current node as an integer. This limit can be configured at startup by use of the [+P] system flag (see the [erl(1)] documentation)

`procs` Returns a binary containing a string of process and port information formatted as in Erlang crash dumps. For more information see the [”How to interpret the Erlang crash dumps”] chapter in the ERTS User's Guide.

`system_version` Returns a string containing the emulator type and version as well as some important properties such as the size of the thread pool, etc.

`system_architecture` Returns a string containing the processor and OS architecture the emulator is built for.

`threads` Returns `true` if the emulator has been compiled with thread support; otherwise, `false` is returned.

`thread_pool_size` Returns the number of threads used for driver calls as an integer.

`trace_control_word` Returns the value of the node's trace control word. For more information see documentation of the [get_tcw] function in the [match specification] documentation in the ERTS User's Guide.

`version` Returns a string containing the version number of the emulator.

`wordsize` Returns the word size in bytes as an integer, i.e. on a 32-bit architecture 4 is returned, and on a 64-bit architecture 8 is returned.

> **Note:**
> `erlang:system_info/1` accepts other arguments than those documented above. These arguments have *intentionally* been left undocumented. This either because the undocumented argument is used for implementation of other documented functionality, or is used for testing and debugging of the emulator. *Do not use the undocumented arguments* of `erlang:system_info/1`, they may have undesirable side-effects, and may be changed or removed at any time without prior notice.

Failure: `badarg` if `What` is not either one of the arguments documented above or one of the undocumented arguments.

`erlang:system_monitor(MonitorPid, Options)`

Sets system performance monitoring options. `MonitorPid` is a local pid that will receive system monitor messages, and `Options` is a list of monitoring options:

{`long_gc, Time`} If a garbage collection in the system takes at least `Time` wallclock milliseconds, a message {`monitor,GcPid,long_gc,Info`} is sent to `MonitorPid`. `GcPid` is the pid that was garbage collected and `Info` is a list of two-element tuples describing the result of the garbage collection. One of the tuples is {`timeout, GcTime`} where `GcTime` is the actual time for the garbage collection in milliseconds. The other are the tuples tagged with `heap_size`, `stack_size`, `mbuf_size` and `heap_block_size` from the `gc_start` trace message (see `erlang:trace/3`).

{`large_heap, Size`} If a garbage collection in the system results in the allocated size of a heap being at least `Size` words, a message {`monitor, GcPid, large_heap, Info`} is sent to `MonitorPid`. `GcPid` and `Info` are the same as for `long_gc` above, except that the tuple tagged with `timeout` is not present.

`busy_port` If a process in the system gets suspended because it sends to a busy port, a message {`monitor, SusPid, busy_port, Port`} is sent to `MonitorPid`. `SusPid` is the pid that got suspended when sending to `Port`.

busy_dist_port If a process in the system gets suspended because it sends to a process on a remote node whose inter-node communication was handled by a busy port, a message {monitor, SusPid, busy_port, Port} is sent to MonitorPid. SusPid is the pid that got suspended when sending through the inter-node communication port Port.

Returns the previous system monitor settings just like erlang:system_monitor/0.

> **Note:**
> If a monitoring process gets so large that it itself starts to cause system monitor messages when garbage collecting, the messages will enlargen the process's message queue and probably make the problem worse.
>
> Keep the monitoring process neat and do not set the system monitor limits too tight.

erlang:system_monitor({MonitorPid, Options})

The same as erlang:system_monitor(MonitorPid, Options).

erlang:system_monitor(undefined)

Clears all system monitoring set by erlang:system_monitor/2.

Returns the previous system monitor settings just like erlang:system_monitor/0.

erlang:system_monitor()

Returns the current system monitoring settings set by erlang:system_monitor/2 as {MonitorPid, Options}. The order of the options may be different from the one that was set.

term_to_binary(Term)

This BIF returns the encoded value of any Erlang term and turns it into the Erlang external term format. It can be used for a variety of purposes, for example writing a term to a file in an efficient way, or sending an Erlang term to some type of communications channel not supported by distributed Erlang.

Returns a binary data object which corresponds to an external representation of the Erlang term Term.

term_to_binary(Term, Options)

This BIF returns the encoded value of any Erlang term and turns it into the Erlang external term format. If the Options list contains the atom compressed, the external term format will be compressed. The compressed format is automatically recognized by binary_to_term/1 in R7.

Returnsa binary data object which corresponds to an external representation of the Erlang term Term.

Failure: badarg if Options is not a list or if contains something else than the supported flags (currently only the atom compressed).

`throw(Any)`

A non-local return from a function. If evaluated within a `catch`, `catch` will return the value `Any`.

```
> catch throw({hello, there}).
{hello, there}
```

Failure: `nocatch` if not evaluated within a catch.

`time()`

Returns the tuple `{Hour, Minute, Second}` of the current system time. The time zone correction is implementation-dependent.

```
> time().
{9, 42, 44}
```

`tl(List)`

Returns the tail of `List`, that is the list minus the first element.

```
> tl([geesties, guilies, beasties]).
[guilies, beasties]
```

Allowed in guard tests.

Failure: `badarg` if `List` is the empty list `[]`, or is not a list.

`erlang:trace(PidSpec, How, Flaglist)`

Turns on (if How == `true`) or off (if How == `false`) the trace flags in `Flaglist` for the process or processes represented by `PidSpec`. `PidSpec` is either a pid for a local process, or one of the following atoms:

`existing` All processes currently existing.

`new` All processes that will be created in the future.

`all` All currently existing processes and all processes that will be created in the future.

`Flaglist` can contain any number of the following atoms (the "message tags" refers to the list of messages following below):

`all` All trace flags except `{tracer, Tracer}` and `cpu_timestamp` that are in their nature different than the others.

`send` Traces the messages the process `Pid` sends. Message tags: `send`, `send_to_non_existing_process`.

`'receive'` Traces the messages the process `Pid` receives. Message tags: `'receive'`.

`procs` Traces process related events, for example `spawn`, `link`, `exit`. Message tags: `spawn`, `exit`, `register`, `unregister`, `link`, `unlink`, `getting_linked`, `getting_unlinked`.

`call` Traces function calls to functions that tracing has been enabled for. Use the erlang:trace_pattern/3 [page 123] BIF to enable tracing for functions. Message tags: `call`, `return_from`.

`silent` To be used in conjunction with the `call` trace flag. Sets the call trace message mode for the process `Pid` to *silent*, i.e, the call tracing is active, match specs are executed as normal, but no call trace messages are generated.

The *silent* mode can, of course, be inhibited by executing `erlang:trace/3` without the `silent` flag, but also by a match spec executing the {`silent, false`} function.

`return_to` Traces the actual return of a process from a traced function back to its caller. This return trace only works together with call trace and functions traced with the `local` option to erlang:trace_pattern/3 [page 123]. The semantics is that a message is sent when a call traced function actually returns, i.e., when a chain of tail recursive calls is ended. There will be only one trace message sent per chain of tail recursive calls, why the properties of tail recursiveness for function calls are kept while tracing with this flag. Using `call` and `return_to` trace together makes it possible to know exactly in which function a process executes at any time.

To get trace messages containing return values from functions, use the {`return_trace`} match_spec action instead.

Message tags: `return_to`.

`running` Traces scheduling of processes. Message tags: `in`, `out`.

`garbage_collection` Traces garbage collections of processes. Message tags: `gc_start`, `gc_end`.

`timestamp` Make a time stamp in all trace messages. The time stamp (Ts) is of the same form as returned by `erlang:now()`.

`cpu_timestamp` A global trace flag for the Erlang node that makes all trace timestamps be in CPU time, not wallclock. It is only allowed with `PidSpec==all`. If the host machine operating system does not support high resolution CPU time measurements, `trace/3` exits with `badarg`.

`arity` Instead of {`Mod, Fun, Args`} in call traces, there will be {`Mod, Fun, Arity`}.

`set_on_spawn` Makes any process created by `Pid` inherit the flags of `Pid`, including the `set_on_spawn` flag.

`set_on_first_spawn` Makes the first process created by `Pid` inherit the flags of `Pid` That process does not inherit the `set_on_first_spawn` flag.

`set_on_link` Makes any process linked by `Pid` inherit the flags of `Pid`, including the `set_on_link` flag.

`set_on_first_link` Makes the first process linked to by `Pid` inherit the flags of `Pid`. That process does not inherit the `set_on_first_link` flag.

{`tracer, Tracer`} `Tracer` should be the pid for a local process or the port identifier for a local port. All trace messages will be sent to the given process or port. If this flag is not given, trace messages will be sent to the process that called `erlang:trace/3`.

The effect of combining `set_on_first_link` with `set_on_link` is the same as having `set_on_first_link` alone. Likewise for `set_on_spawn` and `set_on_first_spawn`.

If the `timestamp` flag is not given, the tracing process will receive the trace messages described below. If the `timestamp` flag is given, the first element of the tuple will be `trace_ts` and the timestamp will be in the last element of the tuple.

{`trace, Pid, 'receive', Message`} When the traced `Pid` receives something.

{`trace, Pid, send, Msg, To`} When `Pid` sends a message.

{trace, Pid, send_to_non_existing_process, Msg, To} When Pid sends a message to a non existing process.

{trace, Pid, call, {M,F,A}} When Pid makes a function/BIF call. The return values of calls are never supplied, only the call and its arguments.

{trace, Pid, return_to, {M,F,A}} When Pid returns *to* function {M,F,A}. This message will be sent if both the call and the return_to flags are present and the function is set to be traced on *local* function calls. The message is only sent when returning from a chain of tail recursive function calls where at least one call generated a call trace message (i.e., the functions match specification matched and {message,false} was not an action).

{trace, Pid, return_from, {M,F,A}, ReturnValue} When Pid returns *from* the function {M,F,A} This trace message is sent when the call flag has been specified, and the function has a match specification with a return_trace action.

{trace, Pid, spawn, Pid2, {M, F, A}} When Pid spawns a new process Pid2. {M, F, A} are the initial function call with arguments for the new process.

Note that A is supposed to be the argument list, but may be any term in the case of an erroneous spawn.

{trace, Pid, exit, Reason} When Pid exits with reason Reason.

{trace, Pid, link, Pid2} When Pid links to a process Pid2.

{trace, Pid, unlink, Pid2} When Pid removes the link from a process Pid2.

{trace, Pid, getting_linked, Pid2} When Pid gets linked to a process Pid2.

{trace, Pid, getting_unlinked, Pid2} When Pid gets unlinked from a process Pid2.

{trace, Pid, register, Name} When Pid gets the name Name registered.

{trace, Pid, unregister, Name} When Pid gets the name Name unregistered. Note that this is done automatically when a registered process exits.

{trace, Pid, in, {M,F,A} | 0} When Pid is scheduled to run. The process will run in function {M,F,A}, where A is always the arity. On some rare occasions the current function cannot be determined, then the last element is 0.

{trace, Pid, out, {M,F,A} | 0} When Pid is scheduled out. The process was running in function {M,F,A} where A is always the arity. On some rare occasions the current function cannot be determined, then the last element is 0.

{trace, Pid, gc_start, Info} Sent when garbage collection is about to be started. Info is a list of two-element tuples, where the first element is a key, and the second is the value. You should not depend on the tuples have any defined order. Currently, the following keys are defined.

heap_size The size of the used part of the heap.

old_heap_size The size of the used part of the old heap.

stack_size The actual size of the stack.

recent_size The size of the data that survived the previous garbage collection.

mbuf_size The combined size of message buffers associated with the process.

All sizes are in words.

{trace, Pid, gc_end, Info} Sent when garbage collection is finished. Info contains the same kind of list as in the gc_start message, but the sizes reflect the new sizes after garbage collection.

If the tracing process dies, the flags will be silently removed.

Only one process can trace a particular process. For this reason, attempts to trace an already traced process will fail.

Returns: A number indicating the number of processes that matched `PidSpec`. If `PidSped` is a pid, the return value will be `1`. If `PidSpec` is `all` or `existing` the return value will be the number of processes running, excluding tracer processes. If `PidSpec` is `new`, the return value will be `0`.

Failure: `badarg` if bad arguments are given. Or if arguments are not supported, for example `cpu_timestamp` is not supported on all platforms.

`erlang:trace_info(PidOrFunc, Item)`

Returns trace information about a process or function.

To get information about a process, `PidOrFunc` should be a pid or the atom `new`. The atom `new` means that the default trace state for processes to be created will be returned. `Item` must have one of the following values:

`flags` Return a list of atoms indicating what kind of traces is enabled for the process. The list will be empty if no traces are enabled, and one or more of the followings atoms if traces are enabled: `send`, `'receive'`, `set_on_spawn`, `call`, `return_to`, `procs`, `set_on_first_spawn`, `set_on_link`, `running`, `garbage_collection`, `timestamp`, and `arity`. The order is arbitrary.

`tracer` Return the identifier for process or port tracing this process. If this process is not being traced, the return value will be `[]`.

To get information about a function, `PidOrFunc` should be a three-element tuple: `{Module, Function, Arity}` or the atom `on_load`. No wildcards are allowed. Return `undefined` if the function does not exist and `false` if the function is not traced at all. `Item` must have one of the following values:

`traced` Return `global` if this function is traced on global function calls, `local` if this function is traced on local function calls (i.e local and global function calls), and `false` if neither local nor global function calls are traced.

`match_spec` Return the match specification for this function, if it has one. If the function is locally or globally traced but has no match specification defined, the returned value is `[]`.

`meta` Return the meta trace tracer process or port for this function, if it has one. If the function is not meta traced the returned value is `false`, and if the function *is* meta traced but has once detected that the tracer proc is invalid, the returned value is `[]`.

`meta_match_spec` Return the meta trace match specification for this function, if it has one. If the function is meta traced but has no match specification defined, the returned value is `[]`.

`call_count` Return the call count value for this function or `true` for the pseudo function `on_load` if call count tracing is active. Return `false` otherwise. See also erlang:trace_pattern/3 [page 123].

`all` Return a list contaning the `{Item, Value}` tuples for all other items, or return `false` if no tracing is active for this function.

The actual return value will be {Item, Value}, where Value is the requested information as described above. If a pid for a dead process was given, or the name of a non-existing function, Value will be undefined.

If PidOrFunc is the on_load, the information returned refers to the default value for code that will be loaded.

erlang:trace_pattern(MFA, MatchSpec)

The same as erlang:trace_pattern(MFA, MatchSpec, []), retained for backward compatibility.

erlang:trace_pattern(MFA, MatchSpec, FlagList)

This BIF is used to enable or disable call tracing for exported functions. It must be combined with erlang:trace/3 [page 119] to set the call trace flag for one or more processes.

Conceptually, call tracing works like this: Inside the Erlang virtual machine there is a set of processes to be traced and a set of functions to be traced. Tracing will be enabled on the intersection of the set. That is, if a process included in the traced process set calls a function included in the traced function set, the trace action will be taken. Otherwise, nothing will happen.

Use erlang:trace/3 [page 119] to add or remove one or more processes to the set of traced processes. Use erlang:trace_pattern/2 to add or remove exported functions to the set of traced functions.

The erlang:trace_pattern/3 BIF can also add match specifications to an exported function. A match specification comprises a pattern that the arguments to the function must match, a guard expression which must evaluate to true and action to be performed. The default action is to send a trace message. If the pattern does not match or the guard fails, the action will not be executed.

The MFA argument should be a tuple like {Module, Function, Arity} or the atom on_load (described below). It can be the module, function, and arity for an exported function (or a BIF in any module). The '_' atom can be used to mean any of that kind. Wildcards can be used in any of the following ways:

{Mod,Func,'_'} All exported functions of any arity named Func in module Mod.

{Mod,'_','_'} All exported functions in module Mod.

{'_','_','_'} All exported functions in all loaded modules.

Other combinations, such as {Mod,'_',Arity}, are not allowed. Local functions will match wildcards only if the local option is in the FlagList.

If the MFA argument is the atom on_load, the match specification and flag list will be used on all modules that are newly loaded.

The MatchSpec argument can take any of the following forms:

false Disable tracing for the matching function(s). Any match specification will be removed.

true Enable tracing for the matching function(s).

MatchSpecList A list of match specifications. An empty list is equivalent to true. See the ERTS User's Guide for a description of match specifications.

`restart` For the `FlagList` option `call_count`: restart the existing counters. The behaviour is undefined for other `FlagList` options.

`pause` For the `FlagList` option `call_count`: pause the existing counters. The behaviour is undefined for other `FlagList` options.

The `FlagList` parameter is a list of options. The following options are allowed:

`global` Turn on or off call tracing for global function calls (i.e., calls specifying the module explicitly). Only exported functions will match and only global calls will generate trace messages. This is the default.

`local` Turn on or off call tracing for all types of function calls. Trace messages will be sent whenever any of the specified functions are called, regardless of how they are called. If the `return_to` flag is set for the process, a `return_to` message will also be sent when this function returns to its caller.

`meta | {meta, Pid}` Turn on or off meta tracing for all types of function calls. Trace messages will be sent to the tracer process or port `Pid` whenever any of the specified functions are called, regardless of how they are called. If no `Pid` is specified, `self()` is used as a default tracer process.

Meta tracing traces all processes and does not care about the process trace flags set by `trace/3`, the trace flags are instead fixed to `[call, timestamp]`.

The match spec function `{return_trace}` works with meta trace and send its trace message to the same tracer process.

`call_count` Starts (`MatchSpec == true`) or stops (`MatchSpec == false`) call count tracing for all types of function calls. For every function a counter is incremented when the function is called, in any process. No process trace flags need to be activated.

If call count tracing is started while already running, the count is restarted from zero. Running counters can be paused with `MatchSpec == pause`. Paused and running counters can be restarted from zero with `MatchSpec == restart`.

The counter value can be read with erlang:trace_info/2 [page 122].

The `global` and `local` options are mutually exclusive and `global` is the default (if no options are specified). The `call_count` and `meta` options perform a kind of local tracing, and can also not be combined with `global`. A function can be *either* globally or locally traced. If global tracing is specified for a specified set of functions; local, meta and call count tracing for the matching set of local functions will be disabled, and vice versa.

When disabling trace, the option must match the type of trace that is set on the function, so that local tracing must be disabled with the `local` option and global tracing with the `global` option (or no option at all), and so forth.

There is no way to directly change part of a match specification list. If a function has a match specification, you can replace it with a completely new one. If you need to change an existing match specification, use the erlang:trace_info/2 [page 122] BIF to retrieve the existing match specification.

Returns the number of exported functions that matched the `MFA` argument. This will be zero if none matched at all.

Failure: `badarg` for invalid `MFA` or `MatchSpec`.

`trunc(Number)`

Returns an integer by the truncation of `Number`.

```
> trunc(5.5).
5
```

Allowed in guard tests.

Failure: badarg if Number is not a number.

tuple_to_list(Tuple)

Returns a list which corresponds to Tuple. Tuple may contain any valid Erlang terms.

```
> tuple_to_list({share, {'Ericsson_B', 163}}).
[share, {'Ericsson_B', 163}]
```

Failure: badarg if Tuple is not a tuple.

erlang:universaltime()

Returns the current date and time according to Universal Time Coordinated (UTC), also called GMT, in the form {{Year, Month, Day}, {Hour, Minute, Second}} if supported by the underlying operating system. If not, erlang:universaltime() is equivalent to erlang:localtime().

```
> erlang:universaltime().
{{1996,11,6},{14,18,43}}
```

erlang:universaltime_to_localtime(DateTime)

Converts UTC date and time in DateTime to local date and time if supported by the underlying operating system. Otherwise, no conversion is done, and DateTime is returned. The return value is of the form {{Year, Month, Day}, {Hour, Minute, Second}}.

```
> erlang:universaltime_to_localtime({{1996,11,6},{14,18,43}}).
{{1996,11,7},{15,18,43}}
```

Failure: badarg if the argument is not a valid date and time tuple {{Year, Month, Day}, {Hour, Minute, Second}}.

unlink(Pid)

Removes a link, if there is one, from the calling process to another process given by the argument Pid.

Returns true. Will not fail if not linked to Pid, or if Pid does not exist.

Failure: badarg if Pid is not a pid.

unregister(Name)

Removes the registered name for a process or port, given by the atom argument Name.

Returns the atom true.

```
> unregister(db).
true
```

Failure: `badarg` if `Name` is not the name of a registered port or process.

Users are advised not to unregister system processes.

`whereis(Name)`

Returns the pid or port identifier registered under `Name` (see `register/2`). Returns `undefined` if no such port or process is registered.

```
> whereis(user).
<0.3.1>
```

Failure: `badarg` if `Name` is not an atom.

`erlang:yield()`

Voluntarily let other processes (if any) get a chance to execute. Using `yield()` is similar to `receive after 1 -> ok end`, except that `yield()` is faster.

error_handler

Erlang Module

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

Exports

`undefined_function(Module, Func, ArgList) -> term()`

Types:

- Module = Func = atom()
- ArgList = [term()]

This function is evaluated if a call is made to `Module:Func(ArgList)` which is undefined. This 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 `Func(ArgList)` call is returned.

Otherwise, it returns, if possible, the value of `apply(Module, Func, ArgList)` after an attempt has been made to autoload `Module`. If this is not possible, the function calling `Module:Func(ArgList)` is exited.

`undefined_lambda(Module, Fun, ArgList) -> term()`

Types:

- Module = Func = atom()
- ArgList = [term()]

This function is evaluated if a call is made to `Fun(ArgList)` when the module defining the `fun` is not loaded. This 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(ArgList)` call is returned.

Otherwise, it returns, if possible, the value of `apply(Fun, ArgList)` after an attempt has been made to autoload `Module`. If this is not possible, the process calling the `fun` is exited.

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 error logger is an event manager behaviour which runs with the registered name `error_logger` (see more about event managers/handlers in the Design Principles chapter and in gen_event(3)). All error messages from the Erlang runtime system are sent to this process as messages with the format {`emulator, Gleader, Str`}, where `Str` is a string which describes the error in plain English. The `Gleader` argument is the group leader process of the process causing the error. This is useful in a distributed setting as all error messages can be returned to the `error_logger` process on the originating node.

Beginning with release R9C of Erlang/OTP, a new class of events, warnings, are added. By default, a warning is similar to an error, i.e. it shows up as an error report in the logs. By using the emulator switch `+W {e|w|i}` one can map warnings to be tagged either as errors (default), warnings (may require rewrite of custom error loggers) or infos.

How warnings are mapped in the running system can be determined by use of the function `error_logger:warning_map/0`, see below.

All errors detected by the standard libraries are reported with the `error_logger` functions. Errors detected in application modules should also be reported through the `error_logger` in order to get uniform reports.

Associated event handlers can be used to add private `types` of reports to the `error_logger`. An event handler which recognizes the specialized report type is first added to the `error_logger` (`add_report_handler/1,2`)

The standard configuration of the `error_logger` supports the logging of errors to the `tty`, or to a specified `file`. There is also a multi-file logger which logs all events, not only the standard error events, to several files. (see log_mf_h(3)).

All error events are tagged with the group leader `Gleader` in order to send the error to the originating node.

Exports

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

Types:

- Pid = pid()
- What = {already_started, Pid} | term()

Starts the `error_logger`. The `start_link` function should be used when the `error_logger` is supervised

```
error_report(Report) -> ok
```

Types:

- Report = [{Tag, Data}] | [term()] | string() | term()
- Tag = term()
- Data = term()

Sends a standard error report event to the error logger. This report event is handled by the standard event handler. The report is formatted as follows:

```
Tag1:   Data1
Tag2:   Data2
Term1
Term2
```

If `Report` is a string(), the string is written.

The report is written with an error heading.

`error_report(Type,Report) -> ok`

Types:

- Type = term()
- Report = [{Tag, Data}] | [term()] | string() | term()
- Tag = term()
- Data = term()

Sends a user defined error report type event to the error logger. If specialized error handling is required, an event handler recognizing this `Type` of report must first be added to the `error_logger`.

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

`info_report(Report) -> ok`

Types:

- Report = [{Tag, Data}] | [term()] | string() | term()
- Tag = term()
- Data = term()

Sends an information report to the error logger. This report event is handled by the standard event handler. The report is formatted as follows:

```
Tag1:   Data1
Tag2:   Data2
Term1
Term2
```

If `Report` is a string(), the string is written.

The report is written with an information heading.

`info_report(Type,Report) -> ok`

Types:

- Type = term()
- Report = [{Tag, Data}] | [term()] | string() | term()
- Tag = term()

- Data = term()

Sends a user defined information report type event to the error logger. If specialized error handling is required, an event handler recognizing this Type of report must first be added to the error_logger.

It is recommended that the Report follows the same structure as info_report/1 above.

```
error_msg(Format) -> ok
error_msg(Format,Args) -> ok
format(Format,Args) -> ok
```

Types:

- Format = string()
- Args = [term()]

Sends an error event to the error logger. The Format and Args arguments are the same as the arguments of io:format/2. These events are handled by the standard event handler.

```
info_msg(Format) -> ok
info_msg(Format,Args) -> ok
```

Types:

- Format = string()
- Args = [term()]

Sends an information event to the error logger. The Format and Args arguments are the same as the arguments of io:format/2. These events are handled by the standard event handler.

```
tty(Flag) -> ok
```

Types:

- Flag = true | false

Enables or disables error printouts to the tty. If Flag is false, all text that the error logger would have sent to the terminal is discarded. If Flag is true, error messages are sent to the terminal screen.

```
logfile(Request) -> ok | FileName | {error, What}
```

Types:

- Request = {open, FileName} | close | filename
- FileName = atom() | string()
- What = term()

This function makes it possible to append a copy of all standard error printouts to a file. It can be used in combination with the tty(false) function in to have a silent system, where all errors are logged to a file.

Request can be:

- {open, Filename}. Opens the file Filename to store a copy of all error messages. Returns ok, or {error, What}.
- close. Closes the current log file. Returns ok, or {error, What}.

- `filename`. Returns {`error, What`} or `FileName`, where `FileName` is the name of the open log file.

There can only be one active log file.

```
add_report_handler(Module) -> ok | Other
add_report_handler(Module,Args) -> ok | Other
```

Types:

- Module = atom()
- Args = term()
- Other = term()

Adds a new event handler to the error logger. The event handler is initialized by a call to the `Module`:init/1 function. This function must return {`ok, State`}. If anything else (`Other`) is returned, the handler is not added.

The report (event) handler will be called for every error event that the error logger receives (`Module:handle_event/2`). Errors dedicated to this handler should be handled accordingly.

```
delete_report_handler(Module) -> Return | {error, What}
```

Types:

- Module = atom()
- Return = term()
- What = term()

Deletes an error report (event) handler. The `Module`:terminate/2 function is called in order to finalize the event handler. The return value of the terminate/2 function is `Return`.

```
swap_handler(ToHandler) -> ok
```

Types:

- ToHandler = tty | {logfile, File}
- File = atom() | string()

The `error_logger` event manager is initially started with a primitive event handler which buffers and prints the raw error events. However, this function does install the standard event handler to be used according to the system configuration.

```
warning_map() -> TagAtom
```

Types:

- TagAtom = {error|warning|info}

This function is used to determine how warnings are mapped in the current system. If `warning_msg/1`, `warning_msg/2`, `warning_report/1` or `warning_report/2` are called, the events might be tagged either as `error`, `warning` or `info`, depending on the switches used when the emulator was started (`+W {i|w|e}`).

```
warning_msg(Format) -> ok
warning_msg(Format,Args) -> ok
```

Types:

- Format = string()
- Args = [term()]

Works like error_msg/1 or error_msg/2 respectively, but the event might be tagged either as error, warning or info, depending on how the emulator was started.

```
warning_report(Report) -> ok
warning_report(Type,Report) -> ok
```

Types:

- Type = term()
- Report = [{Tag, Data}] | [term()] | string() | term()
- Tag = term()
- Data = term()

Works like error_report/1 or error_report/2 respectively, but the event might be tagged either as error, warning or info, depending on how the emulator was started. See description of warnings in the introduction above.

Events

The error logger event manager forwards the following events to all added event handlers. In the events that follow, `Gleader` is the group leader process identity of the error reporting process, and `EPid` is the process identity of the `error_logger`. All other variables are described with the function in which they appear.

`{error_report, Gleader, {Epid, std_error, Report}}` This event is generated when the `error_report/1` function is called.

`{error_report, Gleader, {Epid, Type, Report}}` This event is generated when the `error_report/2` function is called.

`{info_report, Gleader, {Epid, std_info, Report}}` This event is generated when the `info_report/1` function is called.

`{info_report, Gleader, {Epid, Type, Report}}` This event is generated when the `info_report/2` function is called.

`{error, Gleader, {EPid, Format, Args}}` This event is generated when the `error_msg` or `format` functions are called.

`{info_msg, Gleader, {EPid, Format, Args}}` This event is generated when the `info_msg` functions are called.

`{info, Gleader, {EPid, term(), []}}` This structure is only used by the `init` process for erroneously received messages.

`{warning_msg, Gleader, {EPid, Format, Args}}` This structure only appears when the emulator is started with the `+W w` switch, as a result of someone calling `warning_msg/1` or `warning_msg/2`.

`{warning_report, Gleader, {Epid, Type, Report}}` This structure only appears when the emulator is started with the `+W w` switch, as a result of someone calling `warning_report/1` or `warning_report/2`.

> Note:
> All events issued by a process which has the group leader `Gleader` process located on another node will be passed to this node by the `error_logger`.

See Also

gen_event(3), log_mf_h(3)

file

Erlang Module

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

Most functions have a name argument such as a file name or directory name, which is either an atom, a string, or a deep list of characters, lists, and atoms. The `filename` module accepts filenames in the same format.

A path is a list of directory names. If the functions are successful, they return `ok`, or `{ok, Value}`.

If an error occurs, the return value has the format `{error, Reason}`. `Reason` is an atom which is named from the Posix error codes used in Unix, and in the runtime libraries of most C compilers. In the following descriptions of functions, the most typical error codes are listed. By matching the error code, applications can use this information for error recovery. To produce a readable error string, use `format_error/1`.

On operating systems with thread support (Solaris and Windows), 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 option `+A` in the manual page for `erl`.

Exports

`change_group(Filename, Gid)`

Change group of a file. See `write_file_info/2`.

`change_owner(Filename, Uid)`

Change owner of a file. See `write_file_info/2`.

`change_owner(Filename, Uid, Gid)`

Change owner and group of a file. See `write_file_info/2`.

`change_time(Filename, Mtime)`

Change the modification and access times of a file. See `write_file_info/2`.

`change_time(Filename, Mtime, Atime)`

Change the modification and access times of a file. See `write_file_info/2`.

`close(IoDevice)`

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)

Opens file Filename and reads all the Erlang terms in it. Returns one of the following:

{ok, TermList} The file was successfully read.

{error, Atom} An error occurred when opening the file or reading it. The Atom is a Posix error code. See the description of 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 the format_error/1 function to convert the three-element tuple to an English description of the error.

copy(Source, Destination)

Copies the contents of Source to Destination. Source and Destination are either filenames or open file references from e.g open/2.

The same as copy/3 but with infinite byte count.

copy(Source, Destination, ByteCount)

Copies ByteCount bytes from Source to Destination. Source and Destination are either filenames or open file references from e.g open/2.

If Source is a tuple {Filename, ModeList} where ModeList is a mode list as for open/2, the source is opened with read mode prepended to the mode list before the copy, and closed when done.

If Source is a filename, it is interpreted as {Source, []}. The file is thereby read from the beginning.

If Destination is a tuple {Filename, ModeList} where ModeList is a mode list as for open/2, the destination is opened with write mode prepended to the mode list before the copy, and closed when done.

If Destination is a filename, it is interpreted as {Destination, []}. This implies that the previous file contents are overwritten.

If both Source and Destination are filenames or {Filename, ModeList} tuples, the files are opened with [raw, read, binary] and [raw, write, binary] prepended to their mode lists, respectively, to optimize the copy.

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(DirName)

Tries to delete the directory DirName. 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 DirName.

eexist The directory is not empty.

enoent The directory does not exist.

enotdir A component of DirName 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)

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.

eval(Filename)

Opens the file Filename and evaluates all the expression sequences in it. It returns one of the following:

ok The file was read and evaluated. The actual result of the evaluation is not returned; any expression sequence in the file must be there for its side effect.

{error, Atom} An error occurred when opening the file or reading it. The Atom is a Posix error code. See the description of 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 the format_error/1 function to convert the three-element tuple to an English description of the error.

eval(Filename, Bindings)

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)

> **Note:**
> This function is obsolete. Use read_file_info instead.

Retrieves information about a file. Returns {ok, FileInfo} if successful, otherwise {error, Reason}. FileInfo is a tuple with the following fields:

```
{Size,Type,Access,AccessTime,ModifyTime,UnUsed1,UnUsed2}
```

Size The size of the file in bytes.

Type The type of file which is device, directory, regular, or other.

Access The current system access to the file, which is one of the atoms read, write, read_write, or none.

AccessTime The last time the file was read, shown in the format {Year, Month, Day, Hour, Minute, Second}.

ModifyTime The last time the file was written, shown in the format {Year, Month, Day, Hour, Minute, Second}.

UnUsed1, UnUsed2 These fields are not used, but reserved for future expansion. They probably contain unused.

Typical error reasons: Same as for read_file_info/1.

format_error(ErrorDescriptor)

Given the error reason returned by any function in this module, it returns a descriptive string of the error in English.

get_cwd()

Returns {ok, CurDir}, where CurDir (a string) 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)

Drive should be of the form "Letter:", for example "c:". Returns {ok, CurDir} or {error, Reason}, where CurDir (a string) 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 have no concept of drives.

eacces The drive does not exist.

einval The format of Drive is invalid.

ipread_s32bu_p32bu(IoDevice, Location, MaxSize)

Specialised indirect read function for Dets. Equivalent to `pread/3` of a header from `Location` followed by another `pread/3` of the buffer specified by the header.

> **Warning:**
> This function is not intended to be used by others than Dets. It is therefore not well documented.

`list_dir(DirName)`

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

Typical error reasons are:

`eacces` Missing search or write permissions for `DirName` or one of its parent directories.

`enoent` The directory does not exist.

`make_dir(DirName)`

Tries to create the directory `DirName`. 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 `DirName`.

`eexist` There is already a file or directory named `DirName`.

`enoent` A component of `DirName` does not exist.

`enospc` There is a no space left on the device.

`enotdir` A component of `DirName` is not a directory. On some platforms, `enoent` is returned instead.

`make_link(Existing, New)`

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`} will be 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)`

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} will be returned.

Typical error reasons:

eacces Missing read or write permissions for the parent directories of Existing or New.

eexist new already exists.

enotsup Symbolic links are not supported on this platform.

open(Filename, ModeList)

Opens the file Filename in the mode determined by ModeList. ModeList 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. Otherwise, it is truncated (unless combined with read).

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

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 and write/2 functions.
- 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 This option can only be used if the raw option is specified as well. When specified, read operations on the file using the read/2 function 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.

E.g 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.

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 and write gzip compressed files. 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.

If both read and write are specified, the file is created if it does not exists. It is not truncated if it exists.

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.

A file descriptor is the Pid of the process which handles the file. The file process is linked to the process which originally opened the file. If any process to which the file process is linked terminates, the file will be closed by the file process and the process itself will be terminated. The file descriptor returned from this call can be used as an argument to the I/O functions (see io).

> **Note:**
> In previous versions of file, modes were given as on 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)

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. The file is opened and all the terms in it are read. The function returns one of the following:

{ok, TermList, FullName} The file was successfully read. FullName is the full name of the file which was opened and read.

{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. The Atom is a Posix error code. See the description of 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 the format_error/1 function to convert the three-element tuple to an English description of the error.

path_eval(Path, Filename)

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. The file is opened and all the expression sequences in it are evaluated. The function returns one of the following:

{ok, FullName} The file was read. FullName is the full name of the file which was opened and evaluated.

{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. The Atom is a Posix error code. See the description of 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 the format_error/1 function to convert the three-element tuple to an English description of the error.

path_open(Path, Filename, Mode)

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. The function returns one of the following:

{ok, IoDevice, FullName} The file was opened in the requested mode. IoDevice is a reference to the file and FullName is the full name of the file which was opened.

{error, enoent} Filename was not found in the path.

{error, Reason} There was an error opening Filename.

path_script(Path, Filename)

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. The file is opened, all the expression sequences in it are evaluated and the result value is returned. The function returns one of the following:

{ok, Value, FullName} The file was read. FullName is the full name of the file which was opened and evaluated with the result value Value.

{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. The Atom is a Posix error code. See the description of 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 the format_error/1 function to convert the three-element tuple to an English description of the error.

`path_script(Path, Filename, Bindings)`

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)`

If `Pid` is a pid previously 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.

This function is meant for debugging only.

`position(IoDevice, Location)`

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:

`{bof, Offset}` Absolute offset

`{cur, Offset}` Offset from the current position

`{eof, Offset}` Offset from the end of file

`Integer` The same as `{bof, Integer}`

`bof || cur || eof` The same as above with `Offset` 0.

Typical error reasons are:

`einval` Either the `Location` was illegal, or it evaluated to a negative offset in the file. Note that if the resulting position is a negative value you will get an error but after the call it is undefined where the file position will be.

`pread(IoDevice, [{Location, Number}, ...])`

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 `Data` 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.

`pread(IoDevice, Location, Number)`

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.

`pwrite(IoDevice, [{Location, Bytes}, ...])`

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

pwrite(IoDevice, Location, Bytes)

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.

read(IoDevice, Number)

Reads Number bytes from the file described by IoDevice. This function is the only way to read from a file opened in raw mode (although it works for normally opened files, too). Returns:

{ok, ListOrBinary} 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 the number of bytes requested if the end of the file is reached.

eof eof is returned if the Number was greater than zero and end of file was reached before anything at all could be read.

{error, Reason} A Posix error code will be returned if an error occurred.

Typical error reasons:

ebadf The file is not opened for reading.

read_file(Filename)

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)

Retrieves information about a file. Returns {ok, FileInfo} if successful, otherwise {error, Reason}. FileInfo is a record. Its definition can be found by including file.hrl from the kernel application:

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

The record contains the following fields.

size Size of file in bytes.

type The type of the file which can be device, directory, regular, or other.

`access` The current system access to the file, which is one of the atoms `read`, `write`, `read_write`, or `none`.

`atime` The last (local) time the file was read, in the format {{`Year, Month, Day`}, {`Hour, Minute, Second`}}.

`mtime` The last (local) time the file was written, in the format {{`Year, Month, Day`}, {`Hour, Minute, Second`}}.

`ctime` The interpretation of this time field depends on the operating system. On Unix, it is the last time the file or or the inode was changed. In Windows, it is the create time. The format is {{`Year, Month, Day`}, {`Hour, Minute, Second`}}.

`mode` An integer which gives the file permissions as a 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` Number of links to the file (this will always be 1 for file systems which have no concept of links).

`major_device` An integer which 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` Only valid for character devices on Unix. In all other cases, this field is zero.

`inode` An integer which gives the `inode` number. On non-Unix file systems, this field will be zero.

`uid` An integer which indicates the owner of the file. Will be zero for non-Unix file systems.

`gid` An integer which 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_link(Linkname)`

This function returns {ok,Filename} if Linkname 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 Linkname 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(Filename)

This function works like read_file_info/1, except that if Filename 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 Filename is not a symbolic link, this function returns exactly the same result as read_file_info/1. On platforms that do not support symbolic link, this function is always equvivalent to read_file_info/1.

rename(Source, Destination)

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)

Opens the file Filename, evaluates all the expression sequences in it and returns the result value. It returns one of the following:

{ok, Value} The file was read and evaluated with the value Value.

{error, Atom} An error occurred when opening the file or reading it. The Atom is a Posix error code. See the description of 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 the format_error/1 function to convert the three-element tuple to an English description of the error.

script(Filename, Bindings)

The same as script/1 but the variable bindings Bindings are used in the evaluation. See erl_eval(3) about variable bindings.

set_cwd(DirName)

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

Typical error reasons are:

enoent The directory does not exist.

enotdir A component of DirName is not a directory. On some platforms, enoent is returned.

eacces Missing permission for the directory or one of its parents.

sync(IoDevice)

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.

truncate(IoDevice)

Truncates the file referenced by IoDevice at the current position. Returns ok if successful, otherwise {error, Reason}.

write(IoDevice, Bytes)

Writes Bytes (possibly a deep list of characters, or a binary) to the file described 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).

This function returns ok if successful, and {error, Reason} otherwise.

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, Binary)

Writes the contents of the binary data object Binary to the file Filename. The file is created if it does not exist already. 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, Binary, ModeList)

Same as write_file/2, but allows file open mode flags to be specified in ModeList. Mode flags binary and write are implicit so they should not be used.

See also open/2.

write_file_info(Filename, FileInfo)

Change file information. Returns ok if successful, otherwise {error, Reason}. FileInfo is a record. Its definition can be found by including file.hrl from the kernel application:

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

The following fields are used from the record if they are given.

atime The last (local) time the file was read, in the format {{Year, Month, Day}, {Hour, Minute, Second}}.

mtime The last (local) time the file was written, in the format {{Year, Month, Day}, {Hour, Minute, Second}}.

ctime 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. The format is {{Year, Month, Day}, {Hour, Minute, Second}}.

mode An integer which gives the file permissions as a 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 An integer which indicates the owner of the file. Ignored for non-Unix file systems.

gid An integer which 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 I/O error

eisdir illegal operation on a directory

eloop too many levels of symbolic links

emfile too many open files

emlink too many links

enametoolong 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 is 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 I/O device (see `io`). Therefore when data is written to a normal file, the sending of the data to the file process copies all data that is 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 on my machine:

```
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 functionallyequivalent 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 =< 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 =< 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 available options are described in the setopts/2 [page 173] function in the inet manual page.

The possible {error, Reason} results are described in the inet [page 175] manual page.

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.
```

Exports

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

Types:

- ListenSocket = socket()
- Socket = socket()
- Timeout = integer()
- Reason = atom()

Accepts an incoming connection request on a listen socket. `Socket` must be a socket returned from `listen/1`. If no `Timeout` argument is specified, or it is `infinity`, the `accept` function will not return until a connection has been established or the `ListenSocket` has been closed. If accept returns because the `ListenSocket` has been closed {`error, closed`} is returned. If `Timeout` is specified and no connection is accepted within the given time, `accept` will return {`error, timeout`}.

Packets can be sent to the returned socket using the `send/2` function. Packets sent from the peer will be delivered as messages

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

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

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

Types:

- Socket = socket()
- Reason = atom()

Closes an TCP socket.

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

Types:

- Address = string() | atom() | ip_address()
- Port = Timeout = integer()
- Options = list()
- Socket = socket()
- Reason = atom()

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

{port,Port} Specify which local port number to use.

{fd,Fd} If a socket has somehow been connected without using gen_tcp, use this option to pass in the file descriptor for it and create a Socket for it.

inet6 Set up the socket for IPv6

inet Set up the socket for IPv4

common inet options The common inet options available are described in the setopts/2 [page 173] function in the inet manual page.

Packets can be sent to the returned socket using the send/2 function. Packets sent from the peer will be delivered as messages

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

If the socket was closed the following message is delivered:

```
{tcp_closed, Socket}
```

If an error occurred on the socket the following message is delivered:

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

unless the socket is in passive mode, 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 inet_default_connect_options kernel application variable. See the description in the inet reference page for details.

```
controlling_process(Socket, NewOwner) -> ok | {error, eperm}
```

Types:

- Socket = socket()
- NewOwner = pid()

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

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

Types:

- Port = integer()
- Options = list()
- Socket = socket()
- Reason = atom()

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

If the port number is zero, the `listen` function picks an available port number (use `inet:port/1` to retrieve it); otherwise, the specified port number is used.

The available options are described in the setopts/2 [page 173] function in the `inet` manual page.

Additionally, the following options can be given.

{`backlog, B`} B is an integer >= 0. The backlog value defaults to 5. The backlog value defines the maximum length the queue of pending connections may grow to.

{`ip,IPAddress`} If the host has several network interfaces, this option specifies which one to listen on.

{`fd,Fd`} If a listen socket has somehow been opened without using `gen_tcp`, use this option to pass in the file descriptor for it and create a Socket for it.

`inet6` Set up the socket for IPv6

`inet` Set up the socket for IPv4

The returned socket can only be used in calls to `accept`.

> **Note:**
> The default values for options given to `listen` can be affected by the `inet_default_listen_options` kernel application variable. See the description in the `inet` reference page for details.

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

Types:

- Socket = socket()
- Length = integer()
- Packet = list() | binary()
- Timeout = integer()
- Reason = atom()

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

The `Length` argument is only meaningful when the socket is in `raw` mode and denotes number of bytes to read. If `Length` = 0 all available bytes are returned.

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

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

Types:

- Socket = socket()
- Packet = list() | binary()
- Reason = atom()

Sends a packet on a socket.

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

Types:

- Socket = socket()
- Reason = atom()
- How = read | write | read_write

Immediately close a socket in one or two directions.

Closing a socket in the `write` allows you to still read from the socket.

To be able to handle that the peer has done a shutdown on the write side, you'll need to set the `exit_on_close` option for the socket to `false` as described in the setopts/2 [page 173] function in `inet` manual page.

gen_udp

Erlang Module

The gen_udp module is an interface to User Datagram Protocol (UDP).

The possible {error, Reason} results are described in the inet [page 175] manual page.

Exports

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

Types:

- Socket = Reason = term()

Removes the Socket created with open/1 or open/2.

controlling_process(Socket,NewOwner) ->

Types:

- Socket = term()
- NewOwner = pid()

The process associated with a Socket is changed to NewOwner. The NewOwner will receive all subsequent data.

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

Types:

- Port = integer(0..65535)
- Options = list()
- Socket = term()
- Reason = term()

Associates a UDP port number (Port) with the calling process. It returns {ok, Socket}, or {error, Reason}. The returned Socket is used to send packets from this port with the send/4 function. Options is a list of options associated with this port.

When UDP packets arrive at the opened Port they will be delivered as messages of the type {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.

The available options are:

list Received Packet is delivered as a list.

`binary` Received `Packet` is delivered as a binary.

`{ip,IPAddress}` If the host has several network interfaces, this option specifies which one to use.

`{fd,Fd}` If a UDP socket has somehow been opened without using `gen_udp`, use this option to pass in the file descriptor for it and create a Socket for it.

`inet6` Set up the socket for IPv6

`inet` Set up the socket for IPv4

common `inet` options The common `inet` options available are described in the setopts/2 [page 173] function in the `inet` manual page. Default value for the receive buffer option is `{recbuf, 8192}`.

If you set Port to 0, the underlying Operating System assigns a free UDP port. (You can find out which port by calling `inet:port(Socket)`.)

If any of the following functions are called with a `Socket` that was not opened by the calling process, they will return `{error,not_owner}`. The ownership of a `Socket` can be transferred to another process with `controlling_process/2`.

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

Types:

- Socket = socket()
- Address = { integer(), integer(), integer(), integer()}
- Port = integer(0..65535)
- Length = integer()
- Packet = list() | binary()
- Timeout = integer()
- Reason = atom()

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`.

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

Types:

- Address = { integer(), integer(), integer(), integer()} | atom() | string()
- Port = integer(0..65535)
- Packet = [byte()] | binary()
- Reason = term()

Sends `Packet` to the specified address (address, port). It returns `ok`, or `{error, Reason}`. `Address` can be an IP address expressed as a tuple, for example `{192, 0, 0, 1}`. It can also be a host name expressed as an atom or a string, for example 'somehost.some.domain'. `Port` is an integer, and `Packet` is either a list of bytes or a binary.

global

Erlang Module

This documentation describes the Global module which consists of the following functionalities:
1. Registration of global names
2. Global locks
3. Monitoring nodes
4. Maintenance of the fully connected network

These services are controlled via the process `global` which exists on every node. `global` is started automatically when a node is started.

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 are implemented, but for a network of Erlang nodes. A registered name is an alias for a process identity `Pid`. The system 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 extremely quick because it is never a network operation. 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 of a process. 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 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. The 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` server on `N1` then `N3` will make sure that also `N1` and `N3` are connected. If this is not desired, the command line flag `-connect_all false` must be passed to `init` at boot time. In this case, the name registration facility cannot be used (but the lock mechanism will still work.)

Exports

```
del_lock(Id)
del_lock(Id, Nodes) -> void()
```

Types:

- Id = {ResourceId, LockRequesterId}
- ResourceId = term()
- LockRequesterId = term()
- Nodes = [node()]

Deletes the lock `Id` synchronously.

```
notify_all_name(Name, Pid1, Pid2) -> none
```

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) -> Pid1 | Pid2
```

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) -> Pid1 | Pid2
```

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)
register_name(Name, Pid, Resolve) -> yes | no
```

Types:

- Name = term()
- Pid = Pid()
- Resolve = {M, F} where M:F(Name, Pid, Pid2) -> Pid | Pid2 | none

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. This function blocks the global name server during its execution. 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`, `random_notify_name` and `notify_all_name`. 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 a process with a name that is already in use.

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

Types:

- Name = term()
- Pid = Pid()
- Resolve = {M, F} where M:F(Name, Pid, Pid2) -> Pid | Pid2 | none

Atomically changes the registered name `Name` on all nodes to refer to `Pid`.

The `Resolve` function has the same behavior as in `register_name`.

```
send(Name, Msg) -> Pid
```

Types:

- Name = term()
- Msg = term()
- Pid = Pid()

Sends the message `Msg` to the globally registered process `Name`. If `Name` is not a globally registered name, the calling function will exit with reason {`badarg`, {`Name`, `Msg`}}.

```
set_lock(Id)
set_lock(Id, Nodes)
set_lock(Id, Nodes, Retries) -> boolean()
```

Types:

- Id = {ResourceId, LockRequesterId}
- ResourceId = term()
- LockRequesterId = term()
- Nodes = [node()]
- Retries = int() > 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.

`global` keeps track of all processes sharing the same lock, i.e. 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.

```
start()
start_link() -> {ok, Pid} | {error, Reason}
```

This function starts the global name server. Normally, the server is started automatically.

```
stop() -> void()
```

Stops the global name server.

```
sync() -> void()
```

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.

```
trans(Id, Fun)
trans(Id, Fun, Nodes)
trans(Id, Fun, Nodes, Retries) -> Res | aborted
```

Types:

- Id = {ResourceId, LockRequesterId}
- ResourceId = term()
- LockRequesterId = term()
- Fun = fun() | {M, F}
- Nodes = [node()]
- Retries = int() > 0 | infinity
- Res = term()

Sets a lock on `Id` (using `set_lock/3`). Evaluates `Res = Fun()` if successfully locked and returns `Res`. 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) -> void()`

Types:

- Name = term()

Globally removes `Name` from the network of Erlang nodes.

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

Types:

- Name = term()

Returns either an atom `undefined`, or a Pid which is globally associated with `Name`.

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 global_groups-key in the .config file defines the global groups:
{global_groups, [GroupTuple]}

Types:

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

A GroupTuple without PublishType is the same as a GroupTuple with PublishType equal to normal.

The command erl -config File starts a node with a configuration file named File.config. If the global_groups-key is not defined the system will start as a whole, without any partitions. When the key is not defined, the services of this function will not give any extra value to global(3).

A hidden node will establish hidden connections to nodes not part of the same global group, and normal (visible) connections to nodes part of the same global group. Hidden connections aren't published on neither of the connected nodes, i.e. neither of the connected nodes are part of the result from nodes/0 on the other node.

In a hidden global group (a global group defined with PublishType equal to hidden) all nodes are hidden nodes.

Hidden nodes can also be part of normal global groups. Nodes started with the -hidden switch will be hidden nodes even if they are part of a normal group, see erl(1). Other nodes in the group will not be affected by this.

For the processes and nodes to run smoothly using this function the following criteria must be met:

- The global group function must have a server process, global_group, running on each node.
  NOTE: The processes are automatically started and synchronized when a node is started.

- All processes must agree with the group definition in the immediate global group. If two nodes do not agree, these nodes will not synchronize their name space and an error message will be logged in the error logger.
  Example: If one node has an illegal global group definition, such a node will run isolated from the other nodes regarding the global name space; but not regarding other system functions, e.g distribution of applications, refer to chapter NOTE below.
- Nodes can only belong to one global group.

When the global group definitions are to be changed in a system upgrade, the upgrade must fulfill the following steps:

1. First, all nodes which are to be removed from a global group must be taken down.
2. Nodes which are not affected by the redefinition of the global groups are to be upgraded to be aware of the new global group definitions.
   NOTE: All nodes in the system, also nodes in unchanged global groups, must be upgraded. This because e.g `send` must have an accurate view of the total system.
3. Finally, all nodes which are new to a global group can be started.

When a non partitioned system is to be upgraded to become a partitioned system, all nodes belonging to a global group will be disconnected from all nodes not belonging to its immediate global group.

Exports

```
global_groups() -> {OwnGroupName, [OtherGroupName]} | undefined
```

Types:

- OwnGroupName = atom()
- OtherGroupName = atom()
- ErrorMsg = term()

Returns the names of all the global groups known to the immediate global group.

```
info() -> [{state, State}, {own_group_name, atom()}, {own_group_nodes, [Node]},
           {synced_nodes, [Node]}, {sync_error, [Node]}, {no_contact, [Node]},
           {other_groups, Other_grps}, {monitoring, [pid()]}]
```

Types:

- State = no_conf | synced
- Other_grps = [{OtherGrpName, [Node]}]
- OtherGrpName = atom()
- Node = atom()

Returns the state of the global group process. In the following 'nodes' refers to nodes in the immediate global group. `synced_nodes` lists the nodes this node is synchronized with at this moment. lists the nodes defining the own global group. `sync_error` lists the nodes with this node could not be synchronize. `no_contact` lists nodes with this node do not yet have established contact. `other_groups` shows the definition of the other global groups in the system. `monitoring` lists the processes which have subscribed on `nodeup` and `nodedown` messages.

```
monitor_nodes(Flag) -> ok
```

Types:

- Flag = bool()

The requesting process receives {nodeup,Node} and {nodedown,Node} messages about the nodes from the immediate global group. If the flag Flag is set to true the service is turned on; false turns it off.

```
own_nodes() -> [Node] | {error, ErrorMsg}
```

Types:

- Node = atom()
- ErrorMsg = term()

Returns the names of all nodes from the immediate global group, despite of the status of the nodes. Use info/0 to get the information of the current status of the nodes.

```
registered_names({node, Node}) -> [Name] | {error, ErrorMsg}
registered_names({group, GlobalGroupName}) -> [Name]
```

Types:

- Name = term()
- Node = atom()
- GlobalGroupName = atom()
- ErrorMsg = term()

Returns a lists of all globally registered names on the specified node or from the specified global group.

```
send(Name, Msg) -> Pid | {badarg, Msg} | {error, ErrorMsg}
send({node, Node}, Name, Msg) -> Pid | {badarg, Msg} | {error, ErrorMsg}
send({group, GlobalGroupName}, Name, Msg) -> Pid | {badarg, Msg} | {error, ErrorMsg}
```

Types:

- GlobalGroupName = atom()
- Msg = term()
- Name = term()
- Node = atom()
- Pid = pid()
- ErrorMsg = term()

send/2 searches for the registered Name in all global groups defined, in the order of appearance in the .config-file, until the registered name is found or all groups are searched. If Name is found, the message Msg is sent to it. If it is not found, the function exits with reason {badarg, {Name, Msg}}.

send/3 searches for the registered Name in either the specified node or the specified global group. If the registered name is found, the message Msg is sent to that process. If Name is not found, the function exits with reason {badarg, {Name, Msg}}.

```
sync() -> ok
```

sync synchronizes the global name servers on the nodes in the immediate global group. It also unregisters the names registered in the immediate global group on known nodes outside to the immediate global group.

If it the global_groups definition is unvalid, the function exits with reason {error, {'invalid global_groups definition', NodeGrpDef}}.

```
whereis_name(Name) -> Pid | undefined | {error, ErrorMsg}
whereis_name({node, Node}, Name) -> Pid | undefined | {error, ErrorMsg}
whereis_name({group, GlobalGroupName}, Name) -> Pid | undefined | {error, ErrorMsg}
```

Types:

- GlobalGroupName = atom()
- Name = term()
- Node = atom()
- Pid = pid()

whereis_name/1 searches for the registered Name in all global groups defined, in the order of appearance in the .config-file, until the registered name is found or all groups are searched.

whereis_name/2 searches for the registered Name in either the specified node or the specified global group.

Returns either the atom undefined, or the Pid which is associated with Name.

```
start()
start_link() -> {ok, Pid} | {error, Reason}
```

This function starts the global group server. Normally, the server is started automatically.

```
stop() -> void()
```

Stop the global group server.

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

Note also that the send 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) [page 160]

heart

Erlang Module

The heart module 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 (default is 60 seconds) from the last one, 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.

This module is started by the init module during system start-up. The -heart command line flag determines if the heart module should start .

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 erl -heart is started or can be passed on the command line like this: 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).

Exports

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

Types:

- Pid = pid()
- What = void()

Starts the heart program. This function returns ignore if the -heart command line flag is not supplied.

`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.

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

Inet provides access to TCP/IP protocols.

Some functions returns a `hostent` record. Use this line in your module
`-include_lib("kernel/include/inet.hrl").`
to include the record definition.

`h_addr_list` List of addresses for this host

`h_addrtype` Type of address: `inet` or `inet6`

`h_aliases` List of aliases (additional names for host)

`h_length` Length of address in bytes

`h_name` Official name for host

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`}. Addresses returned by any function in the `inet` module will be a tuple.

Hostnames may be specified as either atoms or a strings.

Where an address family is required, valid values are `inet` (standard IPv4 addresses) or `inet6` (IPv6).

Two kernel application variables affect the behaviour of all sockets opened on an erlang node. The kernel application variable `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 application variables 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 delay send would look like this:

```
$ erl -sname test -kernel inet_default_connect_options \n          '[{delay_send
```

Note that the default {`active,true`} currently cannot be changed with these application variables, for internal reasons.

Exports

`get_rc()`

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

`format_error(Tag)`

Types:

- Tag = atom()

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

`gethostbyaddr(Address) -> {ok, Hostent} | {error, Reason}`

Types:

- Address = address()
- Hostent = hostent()

Returns a `hostent` record given an address.

`gethostbyname(Name) -> {ok, Hostent} | {error, Reason}`

Types:

- Hostname = hostname()
- Hostent = hostent()

Returns a `hostent` record given a hostname.

`gethostbyname(Name, Family) -> {ok, Hostent} | {error, Reason}`

Types:

- Hostname = hostname()
- Family = family()
- Hostent = hostent()

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

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

Types:

- Hostname = hostname()

Returns the local hostname. Will never fail.

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

Types:

- Socket = socket()
- Address = address()
- Port = integer()

Returns the local address and port number for a socket.

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

Types:

- Socket = socket()
- Address = address()
- Port = integer()

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

`port(Socket) -> {ok, Number}`

Types:

- Socket = socket()
- Number = integer()

Returns the local port number for a socket.

`close(Socket) -> ok`

Types:

- Socket = socket()

Closes a socket of any type.

`getaddr(IP,inet) -> {ok,{A1,A2,A3,A4}} | {error, Reason}`

Types:

- IP = {A1,A2,A3,A4} | string() | atom()
- A1 = A2 = A3 = A4 = integer()
- Reason = term()

Returns the IP-address as a tuple with integers for `IP` which can be an IP-address a single hostname or a fully qualified hostname. At present only IPv4 adresses (the `inet` argument) is supported, but the function is prepared to support IPv6 (`inet6`) in a near future.

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

Types:

- Socket = term()
- Options = list()

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

{active, Boolean} If the active option is true, which is the default, everything received from the socket will be sent as messages to the receiving process. If the active option is set to false (passive mode), the process must explicitly receive incoming data by calling gen_tcp:recv/N or gen_udp:recv/N (depending on the type of socket). If the active option is set to 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. Note: Active 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. Passive mode or active-once mode provides flow control; the other side will not be able send faster than the receiver can read.

{broadcast, Boolean} Enable/disable permission to send broadcasts. (UDP)

{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 que up the message to be sent whenever the operating system says it can handle it. Setting {delay_send, true} will make *all* messages que 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} Use {dontroute, true} to 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 [page 157] 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 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} If Boolean is true, the TCP_NODELAY option is turned on for the socket, which means that even small amounts of data will be sent immediately. (TCP/IP sockets)

{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; the order of the bytes is big-endian. Each send operation will generate the header, and the header will be stripped off on each receive operation.

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/N` 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.

{`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.

{`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.

{`sndbuf, Integer`} Gives the size of the send buffer to use for the socket.

Please note that the default options for TCP sockets can be changed with the kernel application variables mentioned in the beginning of this document.

ERRORS

The possible error reasons and the corresponding diagnostic strings returned by `format_error/1` are as follows:

`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 proocol 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

init

Erlang Module

`init` is pre-loaded into the system before the system starts and it coordinates the start-up of the system. The first function evaluated at start-up is `boot(Bootargs)`, where `Bootargs` is a list of the arguments supplied to the Erlang runtime system from the local operating system. The Erlang code for the module `init` is always pre-loaded.

`init` reads a boot script which contains instructions on how to initiate the system. The default boot script (`start.boot`) is in the directory `<ERL_INSTALL_DIR>/bin`.

`init` contains functions to fetch command line flags, or arguments, supplied to the Erlang runtime system.

`init` also contains functions to restart, reboot, and stop the system.

Exports

`boot(BootArgs) -> void()`

Types:

- BootArgs = [binary()]

Erlang is started with the command `erl <script-flags> <user-flags>`.

`erl` is the name of the Erlang start-up script. `<script-flags>`, described in erl(1), are read by the script. `<user-flags>` are put into a list and passed as `Args` to `boot/1`.

The `boot/1` function interprets the `boot`, `mode`, and `s` flags. These are described in `COMMAND LINE FLAGS`.

If the `boot` function finds other arguments starting with the character `-`, that argument is interpreted as a flag with zero or more values. It ends the previous argument. For example:

```
erl -run foo bar -charles peterson
```

This starts the Erlang runtime system, evaluates `foo:bar()`, and sets the flag `-charles`, which has the associated value `peterson`.

Other arguments which are passed to the `boot` function, and do not fit into the above description, are passed to the `init` loop as plain arguments.

The special flag `--` can be used to separate plain arguments to `boot` from a preceding flag argument.

The special flag `-extra` causes all following arguments to become plain arguments, and not be subjected to any interpretation by Erlang.

`get_arguments() -> Flags`

Types:

- Flags = [{Flag,FValue}]
- Flag = atom()
- FValue = [Value]
- Value = string()

Returns all flags given to the system.

`get_argument(Flag) -> {ok, Values} | error`

Types:

- Flag = atom()
- Values = [FValue]
- FValue = [Value]
- Value = string()

Returns all values associated with `Flag`. If `Flag` is provided several times, each `FValue` is returned in preserved order.

`get_args() -> [Arg]`

Types:

- Arg = atom()

Returns the additional plain arguments as a list of atoms (possibly empty). It is recommended that `get_plain_arguments/1` be used instead, because of the limited length of atoms.

`get_plain_arguments() -> [Arg]`

Types:

- Arg = string()

Returns the additional plain arguments as a list of strings (possibly empty).

`restart() -> void()`

The system is restarted `inside` the running Erlang node, which means that the emulator is not restarted. All applications are taken down smoothly, all code is unloaded, and all ports are closed before the system is booted again in the same way as initially started. The same `BootArgs` are used again.

To limit the shutdown time, the time `init` is allowed to spend taking down applications, the `-shutdown_time` command line flag should be used.

`reboot() -> void()`

All applications are taken down smoothly, all code is unloaded, and all ports are closed before the Erlang node terminates. If the `-heart` system flag was given, the `heart` program will try to reboot the system. Refer to the `heart` module for more information.

In order to limit the shutdown time, the time `init` is allowed to spend taking down applications, the `-shutdown_time` command line flag should be used.

`stop() -> void()`

All applications are taken down smoothly, all code is unloaded, and all ports are closed before the system terminates. If the `-heart` system flag was given, the `heart` program is terminated before the Erlang node terminates. Refer to the `heart` module for more information.

In order to limit the shutdown time, the time init is allowed to spend taking down applications, the `-shutdown_time` command line flag should be used.

`get_status() -> {InternalStatus, ProvidedStatus}`

Types:

- InternalStatus = starting | started | stopping
- ProvidedStatus = term()

The current status of the `init` process can be inspected. During system start (initialization), `InternalStatus` is `starting`, and `ProvidedStatus` indicates how long the boot script has been interpreted. Each `{progress,Info}` term interpreted in the boot script affects the `ProvidedStatus` status, i.e., `ProvidedStatus` gets the value of `Info`.

`script_id() -> Id`

Types:

- Id = term()

Get the identity of the boot script used to boot the system. `Id` can be any Erlang term. In the delivered boot scripts, `Id` is `{Name,Vsn}`. `Name` and `Vsn` are strings.

Command Line Flags

The `init` module interprets the following flags:

-boot File Specifies the name of the boot script, `File`.boot, used to start the system. Unless `File` contains an absolute path, the system searches for `File`.boot in the current and `<ERL_INSTALL_DIR>/bin` directories

If this flag is omitted, the `<ERL_INSTALL_DIR>/bin/start.boot` boot script is used.

-boot_var Var Directory [Var Directory] If the boot script used contains another path variable than $ROOT, that variable must have a value assigned in order to start the system. A boot variable is used if user applications are installed in a different location than underneath the `<ERL_INSTALL_DIR>/lib` directory. `$Var` is expanded to `Directory` in the boot script.

-mode Mode The mode flag indicates if the system will load code automatically at runtime, or if all code should be loaded during system initialization. `Mode` can be either `interactive` (allow automatic code loading) or `embedded` (load all code during start-up).

-shutdown_time Time Specifies how long time (in ms) the `init` process is allowed to spend shutting down the system. If `Time` milliseconds has elapsed, all processes still existing are *killed*.

If `-shutdown_time` is not specified, the default time is `infinity`.

-run Module [Function [Args]] Evaluate the function during system initialization. `Function` defaults to `start` and `Args` to `[]`. If the function call ends abnormally, the Erlang runtime system stops with an error message.

The arguments after `-run` are used as arguments to Erlang functions. All arguments are passed as strings. For example:

```
erl -run foo -run foo bar -run foo bar baz 1 2
```

This starts the Erlang runtime system and then evaluates the following Erlang functions:

```
foo:start()
foo:bar()
foo:bar(["baz", "1", "2"]).
```

The functions are executed sequentially in the initialization process, which then terminates normally and passes control to the user. This means that a `-run` call which does not terminate will block further processing; to avoid this, use some variant of `spawn` in such cases.

-s Module [Function [Args]] Evaluate the function during system initialization. `Function` defaults to `start` and `Args` to `[]`. If the function call ends abnormally, the Erlang runtime system stops with an error message.

The arguments after `-s` are used as arguments to Erlang functions. All arguments are passed as atoms. For example:

```
erl -s foo -s foo bar -s foo bar baz 1 2
```

This starts the Erlang runtime system and then evaluates the following Erlang functions:

```
foo:start()
foo:bar()
foo:bar([baz, '1', '2']).
```

The functions are executed sequentially in the initialization process, which then terminates normally and passes control to the user. This means that a `-s` call which does not terminate will block further processing; to avoid this, use some variant of `spawn` in such cases.

Due to the 255 character limit on atoms, it is recommended that `-run` be used instead.

-eval Expr Scans, parses and evaluates an arbitrary expression `Expr` during system initialization. If any of these steps fail (syntax error, parse error or crash during evaluation), the Erlang runtime system stops with an error message. Here's an example that seeds the random number generator:

```
$ erl -eval '{X,Y,Z} = now(), random:seed(X, Y, Z).'
```

This example uses Erlang as a hexadecimal calculator:

```
$ erl -noshell -eval 'R = 16#1F+16#A0, io:format("~.16B~n", [R])' -s erlang
BF
```

If multiple -eval expressions are specified, they will be evaluated sequentially in the order specified. -eval expressions are evaluated sequentially with -s and -run function calls (this also in the order specified). As with -s and -run, an evaluation that doesn't terminate, blocks the system initialization process.

-init_debug The `init` process writes some debug information while interpreting the boot script.

Example

```
erl -- a b -children thomas claire -ages 7 3 -- x y
1> init:get_plain_arguments().
["a", "b", "x", "y"]
2> init:get_argument(children).
{ok, [["thomas", "claire"]]}
3> init:get_argument(ages).
{ok, [["7", "3"]]}
4> init:get_argument(silly).
error
```

See also

erl_prim_loader(3) [page 71], heart(3) [page 169]

net_adm

Erlang Module

This module contains various network utility functions.

Exports

`host_file()`

This function reads the `.hosts.erlang` file. It returns the hosts in this file as a list, or it returns `{error, Reason}` if the file cannot be found.

`dns_hostname(Host)`

This function calls epmd for the fully qualified name (DNS) of `Host`. It returns `{ok, Longhostname}` if the call is successful, or `{error, Host}` if `Host` cannot be located by DNS.

`localhost()`

This function returns the fully qualified name of the local host, if it can be found by DNS.

`names(), names(Host)`

This function returns `{ok, List}` or `{error, Reason}`. `List` is a list of tuples on the form `{Name, Port}`. For example: `net_adm:names(elrond) -> {ok,[{"foo",61178},{"ts",61160}]}`.

`ping(Node)`

This function tries to set up a connection to `Node`. It returns `pang` if it fails, and `pong` if it is successful.

`world (), world (verbose)`

This function runs `epmd - names` on all hosts which are specified in the Erlang host file `.hosts.erlang`, collects the replies and then evaluates `ping` on all those nodes. Accordingly, connections are created to all nodes which are running on the hosts specified in the `.hosts.erlang` file. An error message is printed if another user node is found when this is done.

This function can be useful when a node is started, but the names of the other nodes in the network are not initially known.

`world_list (Hostlist), world_list (Hostlist, verbose)`

These functions are the same as world/0 and world/1, but instead of reading the hostfile from .hosts.erlang the hosts are specified in Hostlist.

Files

The .hosts.erlang file consists of a number of host names written as Erlang terms. It can be located in the current work directory, $HOME/.hosts.erlang, or code:root_dir()/.hosts.erlang. 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 examples:

```
'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 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 authentication and monitoring of the network.

An Erlang runtime system can be started from the UNIX command line as follows:

```
% erl -name foobar
```

With this command line, the `net_kernel` is started as `net_kernel:start([foobar])`. See erl(1).

This is done by the system itself, but the `start([Name])` function can also be called directly from the normal Erlang shell prompt, and a normal Erlang runtime system is then converted to a node. The kernel can be shut down with the function `stop()`, but only if the kernel was not started by the system itself. The node is then converted into a normal Erlang runtime system. All other nodes on the network will regard this as a total node crash.

If the system is started as `% erl -sname foobar`, the node name of the node will be `foobar@Host`, where `Host` is the short name of the host (not the fully qualified domain name). The `-name` flag gives a node with the fully qualified domain name. See erl(1).

The system can be started with the flag `-dist_auto_connect` to control automatic connection of remote nodes. See `connect_node/1` below and erl(1).

Exports

`monitor_nodes(Flag, OptionList) -> ok | ignored | Error`

Types:

- Flag = true | false
- OptionList = [Option]
- Option = atom() | {atom(), term()}
- Error = error | {error, term()}

The process evaluating monitor_nodes/2 subscribes or unsubscribes for nodeup/nodedown messages. nodeup messages are delivered to all processes that have subscribed for nodeup/nodedown messages when a new node is connected, and nodedown messages are delivered when a node is disconnected.

If Flag is true, a new subscription is made. If Flag is false, all previous subscriptions with the same OptionList are unsubscribed. Two option lists are considered the same if [lists:usort/1] on the two lists evaluates to terms that are equal.

If OptionList == [] when subscribing, {nodeup, Node}, and {nodedown, Node} messages will be delivered; otherwise, {nodeup, Node, InfoList}, and {nodedown, Node, InfoList} messages will be delivered. Where:

- Node = atom() (the nodename)
- InfoList = [{atom(), term()}]

Also, when OptionList == [] only visible nodes, i.e. nodes that appear in the result of nodes/0 [page 95] and the current node, are monitored.

Currently the following Options are valid:

{node_type, NodeType} Currently valid NodeTypes are:

visible nodeup and nodedown messages will be delivered when visible nodes connect/disconnect. A {node_type, visible} tuple will be part of the InfoList element.

hidden nodeup and nodedown messages will be delivered when hidden nodes connect/disconnect. A {node_type, hidden} tuple will be part of the InfoList element.

all nodeup and nodedown messages will be delivered when a node connect/disconnect. A {node_type, visible} tuple will be part of the InfoList element when the node is visible, and a {node_type, hidden} tuple will be part of the InfoList element when the node is hidden.

nodedown_reason A {nodedown_reason, Reason} tuple will be part of the InfoList element in nodedown messages. Reason can currently 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 No traffic from the connected node during net_ticktime [page 31] seconds.

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.

monitor_nodes(Flag) -> ok | ignored | Error

Types:

- Flag = atom()
- Error = error | {error, term()}

The same as evaluating `net_kernel:monitor_nodes(Flag, [])`.

`allow(NodeList)`

In a simple way, this function limits access to a node from a specific number of named nodes. A node which evaluates this function can only be accessed from nodes listed in the `NodeList` variable. Any access attempts made from nodes not listed in `NodeList` are rejected.

`connect_node(Node)`

Explicitly establishes a connection to the node specified by the atom `Node`. Returns `true` if successful, `false` if not, and `ignored` if `net_kernel` is not started.

This function is only necessary if the system is started with the flag `-dist_auto_connect`. See erl(1).

`set_net_ticktime(NetTicktime, TransitionPeriod) -> Res`

Types:

- NetTicktime = integer() (> 0)
- TransitionPeriod = integer() (>= 0)
- Res = atom() | {atom(), integer()}

Sets `net_ticktime` (see `kernel(6)`) to `NetTicktime` seconds.

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()` (i.e. `((TransitionPeriod*1000 - 1) div MTTI + 1)*MTTI` milliseconds).

If `NetTicktime < PreviousNetTicktime`, the actual `net_ticktime` change will be done at the end of the transition period; otherwise, at the beginning. During the transition period the `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.

Currently defined return values (`Res`):

`unchanged` The `net_ticktime` already had the value of `NetTicktime` and was left unchanged.

`change_initiated` The `net_kernel` has initiated the change of the `net_ticktime` to `NetTicktime` seconds.

`{ongoing_change_to, NewNetTicktime}` The request was *ignored*; because, the `net_kernel` was busy changing the `net_ticktime` to `NewTicktime` seconds.

`set_net_ticktime(NetTicktime) -> Res`

Types:

- NetTicktime = integer()
- Res = atom() | {atom(), integer()}

The same as the call `set_net_ticktime(NetTicktime, 60)`.

`get_net_ticktime() -> Res`

Types:

- Res = integer() | {atom(), integer()}

Gets the `net_ticktime`.

Currently defined return values (`Res`):

`NetTicktime` The `net_ticktime` is `NetTicktime` seconds.

{`ongoing_change_to, NetTicktime`} The `net_kernel` is currently changing the `net_ticktime` to `NetTicktime` seconds.

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 = string() | atom()

Executes `Command` in a command shell of the target OS and returns the 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
```

`find_executable(Name) -> Filename | false`
`find_executable(Name, Path) -> Filename | false`

Types:

- Name = string()
- Path = string()
- 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 (i.e., 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() -> List`

Types:

- List = list() of string

Returns a list of all environement variables. Each environment variable is a single string, containing the name of the variable, followed by =, followed by the 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`.

`type() -> {Osfamily,Osname} | Osfamily`

Types:

- Osfamily = atom() = win32 | unix | vxworks
- Osname = atom()

Returns the `Osfamily` and, in some cases, `Osname` of the current operating system.

On Unix, `Osname` will be same string that `uname -s` returns, but in lower case. For instance, on Solaris 1 and 2, the atom `sunos` will be returned.

In Windows, `Osname` will be either `nt` (on Windows NT), or `windows` (on Windows 95). On VxWorks `Osfamily` alone is returned, i.e. the atom `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() -> {Major, Minor, Release} | VersionString`

Types:

- Major = Minor = Release = integer()
- VersionString = string()

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.

packages

Erlang Module

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) -&gt; 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) -&gt; 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) -&gt; m2:g(X).
g(X) -&gt; 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) -&gt; 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 function (or its abbreviation use/1). E.g.:

```
1&gt; import(foo.bar.m).
ok
2&gt; 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 function `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&gt; 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
```

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 is 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 process 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.

Exports

`create(Name) -> void()`

Types:

- Name = term()

Creates a new, empty process group. The group is globally visible on all nodes. If the group exists, nothing happens.

`delete(Name) -> void()`

Types:

- Name = term()

Deletes a process group.

`get_closest_pid(Name) -> Pid | {error, Reason}`

Types:

- Name = term()

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, Reason}`

Types:

- Name = term()

Returns all processes in the group `Name`. This function should be used from within a client function that accesses the group. It is then optimized for speed.

`get_local_members(Name) -> [Pid] | {error, Reason}`

Types:

- Name = term()

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 then optimized for speed.

`join(Name, Pid) -> ok | {error, Reason}`

Types:

- Name = term()

Joins the process `Pid` to the group `Name`.

`leave(Name, Pid) -> ok | {error, Reason}`

Types:

- Name = term()

Makes the process `Pid` leave the group `Name`.

`which_groups() -> [Name]`

Types:

- Name = term()

Returns a list of all known groups.

`start()`
`start_link() -> {ok, Pid} | {error, Reason}`

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) [page 29], 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.

Exports

`start()`

Starts the `rpc` server. Normally, this is not necessary because the `rpc` server is started automatically.

`stop()`

Stops the `rpc` server.

`call(Node, Module, Function, Args)`

Evaluates `apply(Mod, Fun, Args)` on the node `Node` and returns a value, or `{badrpc, Reason}` if the call fails.

`call(Node, Module, Function, Args, Timeout)`

Evaluates `apply(Mod, Fun, Args)` on the node `Node` and returns a value, or `{badrpc, Reason}` if the call fails. 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.

`cast(Node, Module, Function, Args)`

Causes the expression `apply(Mod, Fun, Args)` to be evaluated on `Node`. No response is delivered and the process which makes the call is not suspended until the evaluation is complete, as is the case with `call/4`. The function immediately returns `true`. Example:

```
> rpc:cast(Node, erlang, halt, [])
```

This function shuts down the node `Node`.

The following function also shuts down the node, but the call returns the tuple `{badrpc, noconnection}`

```
> rpc:call(Node, erlang, halt, [])
```

`block_call(Node, Mod, Fun, Args)`

The `call/4` function causes the server at `Node` to create a new process for each request. This means that several RPCs can be active concurrently. The `rpc` server is not affected if a request does not return a value. 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.

```
> rpc:block_call(Node, erlang, whereis, [file_server]),
```

Returns the Pid of the file server at `Node`.

`block_call(Node, Module, Function, Args, Timeout)`

See `call/4` and `block_call/4`. This is a combination of both. Returns a value, or `{badrpc,Reason}` if the call fails. If the call times out, `Reason` is `timeout`.

`server_call(Node, Name, ReplyWrapper, Msg)`

This function is used when interacting with a server called `Name` at node `Node`. It is assumed that the server receives messages in the format `{From, Request}` and replies in the format `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 `{error, Reason}`, or the answer as produced by the server `Name`.

`abcast(Name, Mess)`

Broadcasts the message `Mess` asynchronously to the registered process `Name` on all nodes, including the current node.

`abcast(Nodes, Name, Mess)`

The same as `abcast/2`, but only to the nodes `Nodes`.

`sbcast(Name, Msg)`

Broadcasts to all nodes synchronously and returns a list of the nodes which have `Name` as a registered server. Returns `{Goodnodes, Badnodes}`.

It is synchronous in the sense that it is known that all servers have received the message when we return from the call. 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 .

`sbcast(Nodes, Name, Msg)`

As `sbcast/2` but only to the nodes in `Nodes`.

`eval_everywhere(Mod, Fun, Args)`

Evaluates the expression `apply(Mod, Fun, Args)` on all nodes. No answers are collected.

`eval_everywhere(Nodes, Mod, Fun, Args)`

Evaluates the expression `apply(Mod, Fun, Args)` on the nodes `Nodes`. No answers are collected.

`multicall(M, F, A)`

The same as `multicall(M, F, A, infinity)`

`multicall(Nodes, M, F, A)`

The same as `multicall(Nodes, M, F, A, infinity)`, where `Nodes` is a list of nodes.

`multicall(M, F, A, Timeout)`

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 `multicall/3` evaluates the expression `apply(M, F, A)` on all nodes and collects the answers. It returns `{Replies, Badnodes}`, where `Badnodes` is a list of the nodes that terminated or timed out during computation, and `Replies` 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
{Replies, _} = rpc:multicall(code, load_binary, [Mod, Bin, File,]),

%% and then maybe check the Replies list.
```

`multicall(Nodes, M, F, A, Timeout)`

Executes the multicall with timeout, but only on the nodes `Nodes`.

`multi_server_call(Name, Msg)`

The function sends Msg to Name on all nodes, and collects the answers. It returns {Replies, Badnodes}, where Badnodes is a list of the nodes which failed during the call. This function assumes that if a request sent to a server called Name, the server replies in the form {Name, node(), Reply}. Otherwise, the function will hang. It also assumes that the server receives messages in the form {From, Msg}, and then replies as From ! {Name, node(), Reply}.

If any of the nodes or servers does not exist or crashes during the call, they appear in the Badnodes list.

> **Warning:**
> If any of the nodes are of an older release of Erlang, the server cannot be monitored, and this function hangs if the server does not exist.

If all nodes are of the current release of Erlang, safe_multi_server_call/2,3 is now obsolete and much more inefficient than multi_server_call/2,3.

The replies are not ordered in any particular way.

`multi_server_call(Nodes, Name, Msg)`

The same as above, but Msg is only sent to Nodes.

`safe_multi_server_call(Name, Msg)`

The same as the multi_server_call/2, except that this function handles the case where the remote node exists, but no server called Name exists there, and the remote node is of an older release of Erlang. This call is also slightly slower than multi_server_call/2 since all request go via the rpc server at the remote sites.

`safe_multi_server_call(Nodes, Name, Msg)`

The same as above, but only on the nodes Nodes.

`async_call(Node, Mod, Fun, Args)`

Call streams with promises is a type of rpc which does not suspend the caller until the result is finished. They return a Key which can be used at a later stage to collect the value. The key can be viewed as a promise to deliver the answer. The expression apply(Mod, Fun, Args) is evaluated for this function on Node. Returns Key which can be used in a subsequent yield/1 (see below).

`yield(Key)`

Delivers the promised answer from a previous async_call operation. If the answer is available, it is returned immediately. Otherwise, the caller of yield/1 is suspended until the answer arrives from Node.

`nb_yield(Key, Timeout)`

This is a non-blocking version of `yield`. It returns the tuple {`value, V`} when the computation has finished, or the atom `timeout` when `Timeout` elapses.

`Timeout` is either a non-negative integer or the atom `infinity`.

`nb_yield(Key)`

Same as `nb_yield(Key, 0)`.

`parallel_eval(ListOfTuples)`

Evaluates the list of size 3 tuples `ListOfTuples`. Each tuple must be of the type {`Mod, Fun, Args`}. Each tuple is sent for evaluation to neighboring nodes, and the replies are collected and returned as a list of individual values. The return values are presented in the same order as the original list `ListOfTuples`.

`pmap({M, F}, Extraargs, List)`

Takes exactly the same arguments and has the same return value as the `lists:map/3` function, except that everything is evaluated in parallel on different nodes.

`pinfo(Pid)`

Location transparent version of `process_info/1`.

`pinfo(Pid, Item)`

Location transparent version of `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 Trace” [page 206] 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.

Exports

`set_token(Component, ComponentValue) -> {Component, PreviousValue}`

Types:

- Component = label | serial | Flag
- Flag = send | 'receive' | print | timestamp
- ComponentValue = FlagValue | LabelValue | SerialValue
- FlagValue = bool() (for Flag)
- LabelValue = integer() (for label)
- SerialValue = {Previous, Current}
- Previous = Current = integer()

Sets the individual `Component` of the trace token to `ComponentValue`. Returns the previous value of the trace token `Component`. The valid `Component, ComponentValue` combinations are:

`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.

`send, bool()` A trace token flag (`true | false`) which enables/disables tracing on message sending. Default is `false`.

`'receive', bool()` A trace token flag (`true | false`) which enables/disables tracing on message reception. Default is `false`.

`print, bool()` A trace token flag (`true | false`) which enables/disables tracing on explicit calls to `seq_trace:print/1`. Default is `false`.

`timestamp, bool()` A trace token flag (`true | false`) which enables/disables a `timestamp` to be generated for each traced event. Default is `false`.

`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(Token) -> PreviousToken`

Types:

- Token = PreviousToken = term() | []

Sets the trace token for the current 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.

`get_token(Component) -> {Component, ComponentValue}`

Types:

- Component = label | serial | Flag
- ComponentValue = FlagValue | LabelValue | SerialValue
- Flag = send | 'receive' | print | timestamp
- FlagValue = bool() (for Flag)
- LabelValue = integer() (for label)
- SerialValue = {Previous, Current} (for serial)
- Previous = Current = integer()

Returns the value of the trace token component`Component`.

`get_token() -> TraceToken`

Types:

- TraceToken = term() | []

Returns the value of the trace token for the current 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.

`print(TraceInfo) -> void`

Types:

- TraceInfo = term()

Puts the Erlang term `TraceInfo` into the sequential trace output if the process currently is executing within a sequential trace and the `print` flag of the trace token is set.

`print(Label, TraceInfo) -> void`

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 of the executing process's sequential trace token.

`reset_trace() -> void`

Sets the trace token to empty for all processes in the 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 Erlang node.

`set_system_tracer(ProcessOrPortId) -> PreviousId`

Types:

- Pid = PreviousId = pid() | port() | false

Sets the system tracer. The system tracer can be either a pid or port denoted by `ProcessOrPortId`. Returns the previous value (which can be `false` if no system tracer is active). The function will generate `{'EXIT',{badarg,Info}}` if `Pid` is not the pid of an existing local process.

`get_system_tracer() -> 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 = integer()
TimeStamp = {Seconds, Milliseconds, Microseconds}
Seconds = Milliseconds = Microseconds = integer()
```

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,Info) and has a trace token with print set to true and label set to Label.

```
Serial = {PreviousSerial, ThisSerial}
PreviousSerial = ThisSerial = integer()
From = To = pid()
```

Serial is a tuple consisting of two integers where the first 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 the 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 distributed 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

1. explicitly by the process itself, through a call to `seq_trace:set_token`, or
2. 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 we call the `Serial` which consists of two integers `Previous` and `Current`. The purpose of `Serial` is 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)`*, the 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 := tprev
  ```

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.

Exports

`start() -> void()`

Starts the basic standard I/O server for the user interface port.

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.

Exports

```
chunk(Continuation)
chunk(Continuation, N) -> {Continuation2, Terms} | {Continuation2, Terms, Badbytes} |
             {Continuation2, eof} | {error, Reason}
```

Types:

- Continuation = continuation()
- N = int() > 0 | infinity
- Continuation2 = continuation()
- Terms= [term()]
- Badbytes = integer()

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
```

Types:

- Continuation = continuation()

This function closes a log file properly.

```
open(Filename) -> OpenRet
open(Filename, N) -> OpenRet
```

Types:

- File = string() | atom()
- N = integer()
- OpenRet = {ok, Continuation} | {error, Reason}
- Continuation = continuation()

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) [page 52]

app

File

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

This file is read by the application controller when an application is loaded. It is also used by the functions in `systools` when generating start scripts etc.

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.

The `.app` file contains one single Erlang term, which is called an *application specification*. The file has the following syntax:

```
{application, Application,
  [{description,  Description},
   {id,           Id},
   {vsn,          Vsn},
   {modules,      [Module1, .., ModuleN]},
   {maxP,         MaxP},
   {maxT,         MaxT},
   {registered,   [Name1, .., NameN]},
   {included_applications, [Appl1, .., ApplN]},
   {applications, [Appl1, .., ApplN]},
   {env,          [{Par1, Val1}, .., {ParN, ValN}]},
   {mod,          {Module, StartArgs}},
   {start_phases, [{Phase1, PhaseArgs1}, .., {PhaseN, PhaseArgsN}]}]}.
```

`Application = atom()` 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,Description}`
  `Description = string()` is a textual description of the application. Defaults to the empty string.
- `{id,Id}`
  `Id = string()` is the product identification of the application. Defaults to the empty string.

- {vsn,Vsn}
  Vsn = string() is the version of the application. Defaults to the empty string.
- {modules,Modules}
  Modules = [atom()] is a list of all the 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. Defaults to the empty list.

  It is also allowed to list a module as {Module,Vsn}, where Vsn = term(). This has no practical implication, however, as there is no check that Vsn is the same value as the vsn attribute of the module. The format is retained for backwards compatibility only.
- {maxP,MaxP}
  Deprecated - will be ignored
  MaxP = int() | infinity is the maximum number of processes allowed in the application. Defaults to infinity.
- {maxT,MaxT}
  MaxT = int() | infinity is the maximum time in milliseconds that the application is allowed to run. After the specified time the application will automatically terminate. Defaults to infinity.
- {registered,Registered}
  Registered = [atom()] is a list of all the names of registered processes started in this application. systools uses this list to detect name clashes between different applications. Defaults to the empty list.
- {included_applications,Applications}
  Applications = [atom()] is a list of all the names of 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. Processes implemented in an included application should be placed underneath a supervisor in the primary application. Defaults to the empty list.
- {applications,Applications}
  Applications = [atom()] is a list of all the names of applications which must be started before this application is 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,Env}
  Env = [{Par,Val}], where Par = atom() and Val = term(), is a list of 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)). Defaults to the empty list.
- {mod,Mod}
  Mod = {Module,StartArgs}, where Module = atom() and StartArgs = term(), specifies the application callback module and the start argument, see application(3).

  The mod key is mandatory for applications implemented as supervision trees, but should be omitted for applications which are code libraries, such as the application STDLIB.
- {start_phases,StartPhases}
  StartPhases = [{Phase,PhaseArgs}], where Phase = atom() and PhaseArgs = term(), is a list of start phases and the attached start arguments for the application. After starting the application, the application master will evaluate the

function `Mod:start_phase(Phase,Type,PhaseArgs)` for each defined start phase, where `Mod` is the callback module as defined by `mod` key. This extended start procedure is intended for synchronized startup of included applications, refer to the chapter about *OTP Design Principles* for more information. Defaults to `undefined`.

SEE ALSO

application(3) [page 33], systools(3)

config

File

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*

Index of Modules and Functions

Modules are typed in *this way*.
Functions are typed in `this way`.