Kernel Application (KERNEL)

version 2.11

Typeset in LaTeX from SGML source using the DOCBUILDER 3.4.1 Document System.

Contents

Kernel Reference Manual

Short Summaries

- Erlang Module user [page ??] – Standard I/O Server
- Erlang Module wrap_log_reader [page ??] – A function to read internally formatted wrap disk logs
- Erlang Module zlib [page ??] – Zlib Compression interface.
- File app [page ??] – Application resource file.
- File config [page ??] – Configuration file.

kernel

No functions are exported.

application

The following functions are exported:

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

- `set_env(Application, Par, Val, Timeout) -> ok`
  [page ??] Set the value of a configuration parameter
- `start(Application) -> ok | {error, Reason}`
  [page ??] Load and start an application
- `start(Application, Type) -> ok | {error, Reason}`
  [page ??] Load and start an application
- `start_type() -> StartType | local | undefined`
  [page ??] Get the start type of an ongoing application startup.
- `stop(Application) -> ok | {error, Reason}`
  [page ??] Stop an application
- `takeover(Application, Type) -> ok | {error, Reason}`
  [page ??] Take over a distributed application
- `unload(Application) -> ok | {error, Reason}`
  [page ??] Unload an application
- `unset_env(Application, Par) -> ok`
  [page ??] Unset the value of a configuration parameter
- `unset_env(Application, Par, Timeout) -> ok`
  [page ??] Unset the value of a configuration parameter
- `which_applications() -> [{Application, Description, Vsn}]`
  [page ??] Get the currently running applications
- `which_applications(Timeout) -> [{Application, Description, Vsn}]`
  [page ??] Get the currently running applications
- `Module:start(StartType, StartArgs) -> {ok, Pid} | {ok, Pid, State} | {error, Reason}`
  [page ??] Start an application
- `Module:start_phase(Phase, StartType, PhaseArgs) -> ok | {error, Reason}`
  [page ??] Extended start of an application
- `Module:prep_stop(State) -> NewState`
  [page ??] Prepare an application for termination
- `Module:stop(State)`
  [page ??] Clean up after termination of an application
- `Module:config_change(Changed, New, Removed) -> ok`
  [page ??] Update the configuration parameters for an application.

auth

The following functions are exported:

- `is_auth(Node) -> yes | no`
  [page ??] Status of communication authorization (deprecated)
- `cookie() -> Cookie`
  [page ??] Magic cookie for local node (deprecated)
- `cookie(TheCookie) -> true`
  [page ??] Set the magic for the local node (deprecated)
- `node_cookie([Node, Cookie]) -> yes | no`
  [page ??] Set the magic cookie for a node and verify authorization (deprecated)
- `node_cookie(Node, Cookie) -> yes | no`
  [page ??] Set the magic cookie for a node and verify authorization (deprecated)

code

The following functions are exported:

- `set_path(Path) -> true | {error, What}`
  [page ??] Set the code server search path
- `get_path() -> Path`
  [page ??] Return the code server search path
- `add_path(Dir) -> true | {error, What}`
  [page ??] Add a directory to the end of the code path
- `add_pathz(Dir) -> true | {error, What}`
  [page ??] Add a directory to the end of the code path
- `add_patha(Dir) -> true | {error, What}`
  [page ??] Add a directory to the beginning of the code path
- `add_paths(Dirs) -> ok`
  [page ??] Add directories to the end of the code path
- `add_pathsz(Dirs) -> ok`
  [page ??] Add directories to the end of the code path
- `add_pathsa(Dirs) -> ok`
  [page ??] Add directories to the beginning of the code path
- `del_path(Name | Dir) -> true | false | {error, What}`
  [page ??] Delete a directory from the code path
- `replace_path(Name, Dir) -> true | {error, What}`
  [page ??] Replace a directory with another in the code path
- `load_file(Module) -> {module, Module} | {error, What}`
  [page ??] Load a module
- `load_abs(Filename) -> {module, Module} | {error, What}`
  [page ??] Load a module, residing in a given file
- `ensure_loaded(Module) -> {module, Module} | {error, What}`
  [page ??] Ensure that a module is loaded
- `load_binary(Module, Filename, Binary) -> {module, Module} | {error, What}`
  [page ??] Load object code for a module
- `delete(Module) -> true | false`
  [page ??] Removes current code for a module
- `purge(Module) -> true | false`
  [page ??] Removes old code for a module
- `soft_purge(Module) -> true | false`
  [page ??] Removes old code for a module, unless no process uses it
- `is_loaded(Module) -> {file, Loaded} | false`
  [page ??] Check if a module is loaded
- `all_loaded() -> [{Module, Loaded}]`
  [page ??] Get all loaded modules
- `which(Module) -> Which`
  [page ??] The object code file of a module
- `get_object_code(Module) -> {Module, Binary, Filename} | error`
  [page ??] Get the object code for a module

- `root_dir() -> string()`
  [page ??] Root directory of Erlang/OTP
- `lib_dir() -> string()`
  [page ??] Library directory of Erlang/OTP
- `lib_dir(Name) -> string() | {error, bad_name}`
  [page ??] Library directory for an application
- `compiler_dir() -> string()`
  [page ??] Library directory for the compiler
- `priv_dir(Name) -> string() | {error, bad_name}`
  [page ??] Priv directory for an application
- `objfile_extension() -> ".beam"`
  [page ??] Object code file extension
- `stick_dir(Dir) -> ok | {error, What}`
  [page ??] Mark a directory as sticky
- `unstick_dir(Dir) -> ok | {error, What}`
  [page ??] Remove a sticky directory mark
- `rehash() -> ok`
  [page ??] Rehash or create code path cache
- `where_is_file(Filename) -> Absname | non_existing`
  [page ??] Full name of a file located in the code path
- `clash() -> ok`
  [page ??] Searche for modules with identical names.

disk_log

The following functions are exported:

- `accessible_logs() -> {[LocalLog], [DistributedLog]}`
  [page ??] Return the accessible disk logs on the current node.
- `alog(Log, Term)`
  [page ??] Asynchronously log an item onto a disk log.
- `balog(Log, Bytes) -> ok | {error, Reason}`
  [page ??] Asynchronously log an item onto a disk log.
- `alog_terms(Log, TermList)`
  [page ??] Asynchronously log several items onto a disk log.
- `balog_terms(Log, BytesList) -> ok | {error, Reason}`
  [page ??] Asynchronously log several items onto a disk log.
- `block(Log)`
  [page ??] Block a disk log.
- `block(Log, QueueLogRecords) -> ok | {error, Reason}`
  [page ??] Block a disk log.
- `change_header(Log, Header) -> ok | {error, Reason}`
  [page ??] Change the head or head_func option for an owner of a disk log.
- `change_notify(Log, Owner, Notify) -> ok | {error, Reason}`
  [page ??] Change the notify option for an owner of a disk log.
- `change_size(Log, Size) -> ok | {error, Reason}`
  [page ??] Change the size of an open disk log.

- `chunk(Log, Continuation)`
  [page ??] Read a chunk of items written to a disk log.
- `chunk(Log, Continuation, N) -> {Continuation2, Terms} | {Continuation2, Terms, Badbytes} | eof | {error, Reason}`
  [page ??] Read a chunk of items written to a disk log.
- `bchunk(Log, Continuation)`
  [page ??] Read a chunk of items written to a disk log.
- `bchunk(Log, Continuation, N) -> {Continuation2, Binaries} | {Continuation2, Binaries, Badbytes} | eof | {error, Reason}`
  [page ??] Read a chunk of items written to a disk log.
- `chunk_info(Continuation) -> InfoList | {error, Reason}`
  [page ??] Return information about a chunk continuation of a disk log.
- `chunk_step(Log, Continuation, Step) -> {ok, Continuation2} | {error, Reason}`
  [page ??] Step forward or backward among the wrap log files of a disk log.
- `close(Log) -> ok | {error, Reason}`
  [page ??] Close a disk log.
- `format_error(Error) -> Chars`
  [page ??] Return an English description of a disk log error reply.
- `inc_wrap_file(Log) -> ok | {error, Reason}`
  [page ??] Change to the next wrap log file of a disk log.
- `info(Log) -> InfoList | {error, no_such_log}`
  [page ??] Return information about a disk log.
- `lclose(Log)`
  [page ??] Close a disk log on one node.
- `lclose(Log, Node) -> ok | {error, Reason}`
  [page ??] Close a disk log on one node.
- `log(Log, Term)`
  [page ??] Log an item onto a disk log.
- `blog(Log, Bytes) -> ok | {error, Reason}`
  [page ??] Log an item onto a disk log.
- `log_terms(Log, TermList)`
  [page ??] Log several items onto a disk log.
- `blog_terms(Log, BytesList) -> ok | {error, Reason}`
  [page ??] Log several items onto a disk log.
- `open(ArgL) -> OpenRet | DistOpenRet`
  [page ??] Open a disk log file.
- `pid2name(Pid) -> {ok, Log} | undefined`
  [page ??] Return the name of the disk log handled by a pid.
- `reopen(Log, File)`
  [page ??] Reopen a disk log and save the old log.
- `reopen(Log, File, Head)`
  [page ??] Reopen a disk log and save the old log.
- `breopen(Log, File, BHead) -> ok | {error, Reason}`
  [page ??] Reopen a disk log and save the old log.
- `sync(Log) -> ok | {error, Reason}`
  [page ??] Flush the contents of a disk log to the disk.

- `truncate(Log)`
  [page ??] Truncate a disk log.
- `truncate(Log, Head)`
  [page ??] Truncate a disk log.
- `btruncate(Log, BHead) -> ok | {error, Reason}`
  [page ??] Truncate a disk log.
- `unblock(Log) -> ok | {error, Reason}`
  [page ??] Unblock a disk log.

erl_boot_server

The following functions are exported:

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

erl_ddll

The following functions are exported:

- `demonitor(MonitorRef) -> ok`
  [page ??] Remove a monitor for a driver
- `info() -> AllInfoList`
  [page ??] Retrieve information about all drivers
- `info(Name) -> InfoList`
  [page ??] Retrieve information about one driver
- `info(Name, Tag) -> Value`
  [page ??] Retrieve specific information about one driver
- `load(Path, Name) -> ok | {error, ErrorDesc}`
  [page ??] Load a driver
- `load_driver(Path, Name) -> ok | {error, ErrorDesc}`
  [page ??] Load a driver
- `monitor(Tag, Item) -> MonitorRef`
  [page ??] Create a monitor for a driver
- `reload(Path, Name) -> ok | {error, ErrorDesc}`
  [page ??] Replace a driver
- `reload_driver(Path, Name) -> ok | {error, ErrorDesc}`
  [page ??] Replace a driver

- `try_load(Path, Name, OptionList) -> {ok,Status} | {ok, PendingStatus, Ref} | {error, ErrorDesc}`
  [page ??] Load a driver
- `try_unload(Name, OptionList) -> {ok,Status} | {ok, PendingStatus, Ref} | {error, ErrorAtom}`
  [page ??] Unload a driver
- `unload(Name) -> ok | {error, ErrorDesc}`
  [page ??] Unload a driver
- `unload_driver(Name) -> ok | {error, ErrorDesc}`
  [page ??] Unload a driver
- `loaded_drivers() -> {ok, Drivers}`
  [page ??] List loaded drivers
- `format_error(ErrorDesc) -> string()`
  [page ??] Format an error descriptor

erl_prim_loader

The following functions are exported:

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

erlang

The following functions are exported:

- `abs(Number) -> int() | float()`
  [page ??] Arithmetical absolute value
- `erlang:append_element(Tuple1, Term) -> Tuple2`
  [page ??] Append an extra element to a tuple
- `apply(Fun, Args) -> term() | empty()`
  [page ??] Apply a function to an argument list
- `apply(Module, Function, Args) -> term() | empty()`
  [page ??] Apply a function to an argument list
- `atom_to_list(Atom) -> string()`
  [page ??] Text representation of an atom
- `binary_to_list(Binary) -> [char()]`
  [page ??] Convert a binary to a list
- `binary_to_list(Binary, Start, Stop) -> [char()]`
  [page ??] Convert part of a binary to a list
- `binary_to_term(Binary) -> term()`
  [page ??] Decode an Erlang external term format binary

- `erlang:fun_to_list(Fun) -> string()`
  [page ??] Text representation of a fun
- `erlang:function_exported(Module, Function, Arity) -> bool()`
  [page ??] Check if a function is exported and loaded
- `garbage_collect() -> true`
  [page ??] Force an immediate garbage collection of the calling process
- `garbage_collect(Pid) -> bool()`
  [page ??] Force an immediate garbage collection of a process
- `get() -> [{Key, Val}]`
  [page ??] Return the process dictionary
- `get(Key) -> Val | undefined`
  [page ??] Return a value from the process dictionary
- `erlang:get_cookie() -> Cookie | nocookie`
  [page ??] Get the magic cookie of the local node
- `get_keys(Val) -> [Key]`
  [page ??] Return a list of keys from the process dictionary
- `erlang:get_stacktrace() -> [{Module, Function, Arity | Args}]`
  [page ??] Get the call stack backtrace of the last exception
- `group_leader() -> GroupLeader`
  [page ??] Get the group leader for the calling process
- `group_leader(GroupLeader, Pid) -> true`
  [page ??] Set the group leader for a process
- `halt()`
  [page ??] Halt the Erlang runtime system and indicate normal exit to the calling environment
- `halt(Status)`
  [page ??] Halt the Erlang runtime system
- `erlang:hash(Term, Range) -> Hash`
  [page ??] Hash function (deprecated)
- `hd(List) -> term()`
  [page ??] Head of a list
- `erlang:hibernate(Module, Function, Args)`
  [page ??] Hibernate a process until a message is sent to it
- `erlang:info(Type) -> Res`
  [page ??] Information about the system (deprecated)
- `integer_to_list(Integer) -> string()`
  [page ??] Text representation of an integer
- `erlang:integer_to_list(Integer, Base) -> string()`
  [page ??] Text representation of an integer
- `iolist_to_binary(IoListOrBinary) -> binary()`
  [page ??] Convert an iolist to a binary
- `iolist_size(Item) -> int()`
  [page ??] Size of an iolist
- `is_alive() -> bool()`
  [page ??] Check whether the local node is alive
- `is_atom(Term) -> bool()`
  [page ??] Check whether a term is an atom

- `is_binary(Term) -> bool()`
  [page ??] Check whether a term is a binary
- `is_boolean(Term) -> bool()`
  [page ??] Check whether a term is a boolean
- `erlang:is_builtin(Module, Function, Arity) -> bool()`
  [page ??] Check if a function is a BIF implemented in C
- `is_float(Term) -> bool()`
  [page ??] Check whether a term is a float
- `is_function(Term) -> bool()`
  [page ??] Check whether a term is a fun
- `is_function(Term, Arity) -> bool()`
  [page ??] Check whether a term is a fun with a given arity
- `is_integer(Term) -> bool()`
  [page ??] Check whether a term is an integer
- `is_list(Term) -> bool()`
  [page ??] Check whether a term is a list
- `is_number(Term) -> bool()`
  [page ??] Check whether a term is a number
- `is_pid(Term) -> bool()`
  [page ??] Check whether a term is a pid
- `is_port(Term) -> bool()`
  [page ??] Check whether a term is a port
- `is_process_alive(Pid) -> bool()`
  [page ??] Check whether a process is alive
- `is_record(Term, RecordTag) -> bool()`
  [page ??] Check whether a term appears to be a record
- `is_record(Term, RecordTag, Size) -> bool()`
  [page ??] Check whether a term appears to be a record
- `is_reference(Term) -> bool()`
  [page ??] Check whether a term is a reference
- `is_tuple(Term) -> bool()`
  [page ??] Check whether a term is a tuple
- `length(List) -> int()`
  [page ??] Length of a list
- `link(Pid) -> true`
  [page ??] Create a link to another process (or port)
- `list_to_atom(String) -> atom()`
  [page ??] Convert from text representation to an atom
- `list_to_binary(IoList) -> binary()`
  [page ??] Convert a list to a binary
- `list_to_existing_atom(String) -> atom()`
  [page ??] Convert from text representation to an atom
- `list_to_float(String) -> float()`
  [page ??] Convert from text representation to a float
- `list_to_integer(String) -> int()`
  [page ??] Convert from text representation to an integer

- `erlang:list_to_integer(String, Base) -> int()`
  [page ??] Convert from text representation to an integer
- `list_to_pid(String) -> pid()`
  [page ??] Convert from text representation to a pid
- `list_to_tuple(List) -> tuple()`
  [page ??] Convert a list to a tuple
- `load_module(Module, Binary) -> {module, Module} | {error, Reason}`
  [page ??] Load object code for a module
- `erlang:loaded() -> [Module]`
  [page ??] List of all loaded modules
- `erlang:localtime() -> {Date, Time}`
  [page ??] Current local date and time
- `erlang:localtime_to_universaltime({Date1, Time1}) -> {Date2, Time2}`
  [page ??] Convert from local to Universal Time Coordinated (UTC) date and time
- `erlang:localtime_to_universaltime({Date1, Time1}, IsDst) -> {Date2, Time2}`
  [page ??] Convert from local to Universal Time Coordinated (UTC) date and time
- `make_ref() -> ref()`
  [page ??] Return an almost unique reference
- `erlang:make_tuple(Arity, InitialValue) -> tuple()`
  [page ??] Create a new tuple of a given arity
- `erlang:md5(Data) -> Digest`
  [page ??] Compute an MD5 message digest
- `erlang:md5_final(Context) -> Digest`
  [page ??] Finish the update of an MD5 context and return the computed MD5 message digest
- `erlang:md5_init() -> Context`
  [page ??] Create an MD5 context
- `erlang:md5_update(Context, Data) -> NewContext`
  [page ??] Update an MD5 context with data, and return a new context
- `erlang:memory() -> [{Type, Size}]`
  [page ??] Information about dynamically allocated memory
- `erlang:memory(Type | [Type]) -> Size | [{Type, Size}]`
  [page ??] Information about dynamically allocated memory
- `module_loaded(Module) -> bool()`
  [page ??] Check if a module is loaded
- `erlang:monitor(Type, Item) -> MonitorRef`
  [page ??] Start monitoring
- `monitor_node(Node, Flag) -> true`
  [page ??] Monitor the status of a node
- `erlang:monitor_node(Node, Flag, Options) -> true`
  [page ??] Monitor the status of a node
- `node() -> Node`
  [page ??] Name of the local node
- `node(Arg) -> Node`
  [page ??] At which node is a pid, port or reference located

- `nodes() -> Nodes`
  [page ??] All visible nodes in the system
- `nodes(Arg | [Arg]) -> Nodes`
  [page ??] All nodes of a certain type in the system
- `now() -> {MegaSecs, Secs, MicroSecs}`
  [page ??] Elapsed time since 00:00 GMT
- `open_port(PortName, PortSettings) -> port()`
  [page ??] Open a port
- `erlang:phash(Term, Range) -> Hash`
  [page ??] Portable hash function
- `erlang:phash2(Term [, Range]) -> Hash`
  [page ??] Portable hash function
- `pid_to_list(Pid) -> string()`
  [page ??] Text representation of a pid
- `port_close(Port) -> true`
  [page ??] Close an open port
- `port_command(Port, Data) -> true`
  [page ??] Send data to a port
- `port_connect(Port, Pid) -> true`
  [page ??] Set the owner of a port
- `port_control(Port, Operation, Data) -> Res`
  [page ??] Perform a synchronous control operation on a port
- `erlang:port_call(Port, Operation, Data) -> term()`
  [page ??] Synchronous call to a port with term data
- `erlang:port_info(Port) -> [{Item, Info}] | undefined`
  [page ??] Information about a port
- `erlang:port_info(Port, Item) -> {Item, Info} | undefined | []`
  [page ??] Information about a port
- `erlang:port_to_list(Port) -> string()`
  [page ??] Text representation of a port identifier
- `erlang:ports() -> [port()]`
  [page ??] All open ports
- `pre_loaded() -> [Module]`
  [page ??] List of all pre-loaded modules
- `erlang:process_display(Pid, Type) -> void()`
  [page ??] Write information about a local process on standard error
- `process_flag(Flag, Value) -> OldValue`
  [page ??] Set process flags for the calling process
- `process_flag(Pid, Flag, Value) -> OldValue`
  [page ??] Set process flags for a process
- `process_info(Pid) -> [{Item, Info}] | undefined`
  [page ??] Information about a process
- `process_info(Pid, Item) -> {Item, Info} | undefined | []`
  [page ??] Information about a process
- `processes() -> [pid()]`
  [page ??] All processes

- `purge_module(Module) -> void()`
  [page ??] Remove old code for a module
- `put(Key, Val) -> OldVal | undefined`
  [page ??] Add a new value to the process dictionary
- `erlang:raise(Class, Reason, Stacktrace)`
  [page ??] Stop execution with an exception of given class, reason and call stack backtrace
- `erlang:read_timer(TimerRef) -> int() | false`
  [page ??] Number of milliseconds remaining for a timer
- `erlang:ref_to_list(Ref) -> string()`
  [page ??] Text representation of a reference
- `register(RegName, Pid | Port) -> true`
  [page ??] Register a name for a pid (or port)
- `registered() -> [RegName]`
  [page ??] All registered names
- `erlang:resume_process(Pid) -> true`
  [page ??] Resume a suspended process
- `round(Number) -> int()`
  [page ??] Return an integer by rounding a number
- `self() -> pid()`
  [page ??] Pid of the calling process
- `erlang:send(Dest, Msg) -> Msg`
  [page ??] Send a message
- `erlang:send(Dest, Msg, [Option]) -> Res`
  [page ??] Send a message conditionally
- `erlang:send_after(Time, Dest, Msg) -> TimerRef`
  [page ??] Start a timer
- `erlang:send_nosuspend(Dest, Msg) -> bool()`
  [page ??] Try to send a message without ever blocking
- `erlang:send_nosuspend(Dest, Msg, Options) -> bool()`
  [page ??] Try to send a message without ever blocking
- `erlang:set_cookie(Node, Cookie) -> true`
  [page ??] Set the magic cookie of a node
- `setelement(Index, Tuple1, Value) -> Tuple2`
  [page ??] Set Nth element of a tuple
- `size(Item) -> int()`
  [page ??] Size of a tuple or binary
- `spawn(Fun) -> pid()`
  [page ??] Create a new process with a fun as entry point
- `spawn(Node, Fun) -> pid()`
  [page ??] Create a new process with a fun as entry point on a given node
- `spawn(Module, Function, Args) -> pid()`
  [page ??] Create a new process with a function as entry point
- `spawn(Node, Module, Function, ArgumentList) -> pid()`
  [page ??] Create a new process with a function as entry point on a given node
- `spawn_link(Fun) -> pid()`
  [page ??] Create and link to a new process with a fun as entry point

- spawn_link(Node, Fun) ->
  [page ??] Create and link to a new process with a fun as entry point on a specified node
- spawn_link(Module, Function, Args) -> pid()
  [page ??] Create and link to a new process with a function as entry point
- spawn_link(Node, Module, Function, Args) -> pid()
  [page ??] Create and link to a new process with a function as entry point on a given node
- erlang:spawn_monitor(Fun) -> {pid(),reference()}
  [page ??] Create and monitor a new process with a fun as entry point
- erlang:spawn_monitor(Module, Function, Args) -> {pid(),reference()}
  [page ??] Create and monitor a new process with a function as entry point
- spawn_opt(Fun, [Option]) -> pid() | {pid(),reference()}
  [page ??] Create a new process with a fun as entry point
- spawn_opt(Node, Fun, [Option]) -> pid()
  [page ??] Create a new process with a fun as entry point on a given node
- spawn_opt(Module, Function, Args, [Option]) -> pid() | {pid(),reference()}
  [page ??] Create a new process with a function as entry point
- spawn_opt(Node, Module, Function, Args, [Option]) -> pid()
  [page ??] Create a new process with a function as entry point on a given node
- split_binary(Bin, Pos) -> {Bin1, Bin2}
  [page ??] Split a binary into two
- erlang:start_timer(Time, Dest, Msg) -> TimerRef
  [page ??] Start a timer
- statistics(Type) -> Res
  [page ??] Information about the system
- erlang:suspend_process(Pid) -> true
  [page ??] Suspend a process
- erlang:system_flag(Flag, Value) -> OldValue
  [page ??] Set system flags
- erlang:system_info(Type) -> Res
  [page ??] Information about the system
- erlang:system_monitor() -> MonSettings
  [page ??] Current system performance monitoring settings
- erlang:system_monitor(undefined | {MonitorPid, Options}) -> MonSettings
  [page ??] Set or clear system performance monitoring options
- erlang:system_monitor(MonitorPid, [Option]) -> MonSettings
  [page ??] Set system performance monitoring options
- term_to_binary(Term) -> ext_binary()
  [page ??] Encode a term to an Erlang external term format binary
- term_to_binary(Term, [Option]) -> ext_binary()
  [page ??] Encode a term to en Erlang external term format binary
- throw(Any)
  [page ??] Throw an exception

- `time() -> {Hour, Minute, Second}`
  [page ??] Current time
- `tl(List1) -> List2`
  [page ??] Tail of a list
- `erlang:trace(PidSpec, How, FlagList) -> int()`
  [page ??] Set trace flags for a process or processes
- `erlang:trace_delivered(Tracee) -> Ref`
  [page ??] Notification when trace has been delivered
- `erlang:trace_info(PidOrFunc, Item) -> Res`
  [page ??] Trace information about a process or function
- `erlang:trace_pattern(MFA, MatchSpec) -> int()`
  [page ??] Set trace patterns for global call tracing
- `erlang:trace_pattern(MFA, MatchSpec, FlagList) -> int()`
  [page ??] Set trace patterns for tracing of function calls
- `trunc(Number) -> int()`
  [page ??] Return an integer by the truncating a number
- `tuple_to_list(Tuple) -> [term()]`
  [page ??] Convert a tuple to a list
- `erlang:universaltime() -> {Date, Time}`
  [page ??] Current date and time according to Universal Time Coordinated (UTC)
- `erlang:universaltime_to_localtime({Date1, Time1}) -> {Date2, Time2}`
  [page ??] Convert from Universal Time Coordinated (UTC) to local date and time
- `unlink(Id) -> true`
  [page ??] Remove a link, if there is one, to another process or port
- `unregister(RegName) -> true`
  [page ??] Remove the registered name for a process (or port)
- `whereis(RegName) -> pid() | port() | undefined`
  [page ??] Get the pid (or port) with a given registered name
- `erlang:yield() -> true`
  [page ??] Let other processes get a chance to execute

error_handler

The following functions are exported:

- `undefined_function(Module, Function, Args) -> term()`
  [page ??] Called when an undefined function is encountered
- `undefined_lambda(Module, Fun, Args) -> term()`
  [page ??] Called when an undefined lambda (fun) is encountered

error_logger

The following functions are exported:

- `error_msg(Format) -> ok`
  [page ??] Send an standard error event to the error logger
- `error_msg(Format, Data) -> ok`
  [page ??] Send an standard error event to the error logger

- `format(Format, Data) -> ok`
  [page ??] Send an standard error event to the error logger
- `error_report(Report) -> ok`
  [page ??] Send a standard error report event to the error logger
- `error_report(Type, Report) -> ok`
  [page ??] Send a user defined error report event to the error logger
- `warning_map() -> Tag`
  [page ??] Return the current mapping for warning events
- `warning_msg(Format) -> ok`
  [page ??] Send a standard warning event to the error logger
- `warning_msg(Format, Data) -> ok`
  [page ??] Send a standard warning event to the error logger
- `warning_report(Report) -> ok`
  [page ??] Send a standard warning report event to the error logger
- `warning_report(Type, Report) -> ok`
  [page ??] Send a user defined warning report event to the error logger
- `info_msg(Format) -> ok`
  [page ??] Send a standard information event to the error logger
- `info_msg(Format, Data) -> ok`
  [page ??] Send a standard information event to the error logger
- `info_report(Report) -> ok`
  [page ??] Send a standard information report event to the error logger
- `info_report(Type, Report) -> ok`
  [page ??] Send a user defined information report event to the error logger
- `add_report_handler(Handler) -> Result`
  [page ??] Add an event handler to the error logger
- `add_report_handler(Handler, Args) -> Result`
  [page ??] Add an event handler to the error logger
- `delete_report_handler(Handler) -> Result`
  [page ??] Delete an event handler from the error logger
- `tty(Flag) -> ok`
  [page ??] Enable or disable printouts to the tty
- `logfile(Request) -> ok | Filename | {error, What}`
  [page ??] Enable or disable error printouts to a file

file

The following functions are exported:

- `change_group(Filename, Gid) -> ok | {error, Reason}`
  [page ??] Change group of a file
- `change_owner(Filename, Uid) -> ok | {error, Reason}`
  [page ??] Change owner of a file
- `change_owner(Filename, Uid, Gid) -> ok | {error, Reason}`
  [page ??] Change owner and group of a file
- `change_time(Filename, Mtime) -> ok | {error, Reason}`
  [page ??] Change the modification time of a file

- `change_time(Filename, Mtime, Atime) -> ok | {error, Reason}`
  [page ??] Change the modification and last access time of a file
- `close(IoDevice) -> ok | {error, Reason}`
  [page ??] Close a file
- `consult(Filename) -> {ok, Terms} | {error, Reason}`
  [page ??] Read Erlang terms from a file
- `copy(Source, Destination) ->`
  [page ??] Copy file contents
- `copy(Source, Destination, ByteCount) -> {ok, BytesCopied} | {error, Reason}`
  [page ??] Copy file contents
- `del_dir(Dir) -> ok | {error, Reason}`
  [page ??] Delete a directory
- `delete(Filename) -> ok | {error, Reason}`
  [page ??] Delete a file
- `eval(Filename) -> ok | {error, Reason}`
  [page ??] Evaluate Erlang expressions in a file
- `eval(Filename, Bindings) -> ok | {error, Reason}`
  [page ??] Evaluate Erlang expressions in a file
- `file_info(Filename) -> {ok, FileInfo} | {error, Reason}`
  [page ??] Get information about a file (deprecated)
- `format_error(Reason) -> Chars`
  [page ??] Return a descriptive string for an error reason
- `get_cwd() -> {ok, Dir} | {error, Reason}`
  [page ??] Get the current working directory
- `get_cwd(Drive) -> {ok, Dir} | {error, Reason}`
  [page ??] Get the current working directory for the drive specified
- `list_dir(Dir) -> {ok, Filenames} | {error, Reason}`
  [page ??] List files in a directory
- `make_dir(Dir) -> ok | {error, Reason}`
  [page ??] Make a directory
- `make_link(Existing, New) -> ok | {error, Reason}`
  [page ??] Make a hard link to a file
- `make_symlink(Name1, Name2) -> ok | {error, Reason}`
  [page ??] Make a symbolic link to a file or directory
- `open(Filename, Modes) -> {ok, IoDevice} | {error, Reason}`
  [page ??] Open a file
- `path_consult(Path, Filename) -> {ok, Terms, FullName} | {error, Reason}`
  [page ??] Read Erlang terms from a file
- `path_eval(Path, Filename) -> {ok, FullName} | {error, Reason}`
  [page ??] Evaluate Erlang expressions in a file
- `path_open(Path, Filename, Modes) -> {ok, IoDevice, FullName} | {error, Reason}`
  [page ??] Open a file

- `path_script(Path, Filename) -> {ok, Value, FullName} | {error, Reason}`
  [page ??] Evaluate and return the value of Erlang expressions in a file
- `path_script(Path, Filename, Bindings) -> {ok, Value, FullName} | {error, Reason}`
  [page ??] Evaluate and return the value of Erlang expressions in a file
- `pid2name(Pid) -> string() | undefined`
  [page ??] Return the name of the file handled by a pid
- `position(IoDevice, Location) -> {ok, NewPosition} | {error, Reason}`
  [page ??] Set position in a file
- `pread(IoDevice, LocNums) -> {ok, DataL} | {error, Reason}`
  [page ??] Read from a file at certain positions
- `pread(IoDevice, Location, Number) -> {ok, Data} | {error, Reason}`
  [page ??] Read from a file at a certain position
- `pwrite(IoDevice, LocBytes) -> ok | {error, {N, Reason}}`
  [page ??] Write to a file at certain positions
- `pwrite(IoDevice, Location, Bytes) -> ok | {error, Reason}`
  [page ??] Write to a file at a certain position
- `read(IoDevice, Number) -> {ok, Data} | eof | {error, Reason}`
  [page ??] Read from a file
- `read_file(Filename) -> {ok, Binary} | {error, Reason}`
  [page ??] Read a file
- `read_file_info(Filename) -> {ok, FileInfo} | {error, Reason}`
  [page ??] Get information about a file
- `read_link(Name) -> {ok, Filename} | {error, Reason}`
  [page ??] See what a link is pointing to
- `read_link_info(Name) -> {ok, FileInfo} | {error, Reason}`
  [page ??] Get information about a link or file
- `rename(Source, Destination) -> ok | {error, Reason}`
  [page ??] Rename a file
- `script(Filename) -> {ok, Value} | {error, Reason}`
  [page ??] Evaluate and return the value of Erlang expressions in a file
- `script(Filename, Bindings) -> {ok, Value} | {error, Reason}`
  [page ??] Evaluate and return the value of Erlang expressions in a file
- `set_cwd(Dir) -> ok | {error,Reason}`
  [page ??] Set the current working directory
- `sync(IoDevice) -> ok | {error, Reason}`
  [page ??] Synchronizes the in-memory state of a file with that on the physical medium
- `truncate(IoDevice) -> ok | {error, Reason}`
  [page ??] Truncate a file
- `write(IoDevice, Bytes) -> ok | {error, Reason}`
  [page ??] Write to a file
- `write_file(Filename, Binary) -> ok | {error, Reason}`
  [page ??] Write a file
- `write_file(Filename, Binary, Modes) -> ok | {error, Reason}`
  [page ??] Write a file
- `write_file_info(Filename, FileInfo) -> ok | {error, Reason}`
  [page ??] Change information about a file

gen_sctp

The following functions are exported:

- `abort(sctp_socket(), Assoc) -> ok | {error, posix()}`
  [page ??] Abnormally terminate the association given by Assoc, without flushing of unsent data
- `close(sctp_socket()) -> ok | {error, posix()}`
  [page ??] Completely close the socket and all associations on it
- `connect(Socket, IP, Port, Opts) -> {ok,Assoc} | {error, posix()}`
  [page ??] Same as `connect(Socket, IP, Port, Opts, infinity)`.
- `connect(Socket, IP, Port, [Opt], Timeout) -> {ok, Assoc} | {error, posix()}`
  [page ??] Establish a new association for the socket `Socket`, with a peer (SCTP server socket)
- `controlling_process(sctp_socket(), pid()) -> ok`
  [page ??] Assign a new controlling process pid to the socket
- `eof(Socket, Assoc) -> ok | {error, Reason}`
  [page ??] Gracefully terminate the association given by Assoc, with flushing of all unsent data
- `listen(Socket, IsServer) -> ok | {error, Reason}`
  [page ??] Set up a socket to listen.
- `open() -> {ok, Socket} | {error, posix()}`
  [page ??] Create an SCTP socket and bind it to local addresses
- `open(Port) -> {ok, Socket} | {error, posix()}`
  [page ??] Create an SCTP socket and bind it to local addresses
- `open([Opt]) -> {ok, Socket} | {error, posix()}`
  [page ??] Create an SCTP socket and bind it to local addresses
- `recv(sctp_socket()) -> {ok, {FromIP, FromPort, AncData, BinMsg}} | {error, Reason}`
  [page ??] Receive a message from a socket
- `recv(sctp_socket(), timeout()) -> {ok, {FromIP, FromPort, AncData, Data}} | {error, Reason}`
  [page ??] Receive a message from a socket
- `send(Socket, SndRcvInfo, Data) -> ok | {error, Reason}`
  [page ??] Send a message using an `#sctp_sndrcvinfo{}` record
- `send(Socket, Assoc, Stream, Data) -> ok | {error, Reason}`
  [page ??] Send a message over an existing association and given stream
- `error_string(integer()) -> ok | string() | undefined`
  [page ??] Translate an SCTP error number into a string

gen_tcp

The following functions are exported:

- `connect(Address, Port, Options) -> {ok, Socket} | {error, Reason}`
  [page ??] Connect to a TCP port

- `connect(Address, Port, Options, Timeout) -> {ok, Socket} | {error, Reason}`
  [page ??] Connect to a TCP port
- `listen(Port, Options) -> {ok, ListenSocket} | {error, Reason}`
  [page ??] Set up a socket to listen on a port
- `accept(ListenSocket) -> {ok, Socket} | {error, Reason}`
  [page ??] Accept an incoming connection request on a listen socket
- `accept(ListenSocket, Timeout) -> {ok, Socket} | {error, Reason}`
  [page ??] Accept an incoming connection request on a listen socket
- `send(Socket, Packet) -> ok | {error, Reason}`
  [page ??] Send a packet
- `recv(Socket, Length) -> {ok, Packet} | {error, Reason}`
  [page ??] Receive a packet from a passive socket
- `recv(Socket, Length, Timeout) -> {ok, Packet} | {error, Reason}`
  [page ??] Receive a packet from a passive socket
- `controlling_process(Socket, Pid) -> ok | {error, eperm}`
  [page ??] Change controlling process of a socket
- `close(Socket) -> ok | {error, Reason}`
  [page ??] Close a TCP socket
- `shutdown(Socket, How) -> ok | {error, Reason}`
  [page ??] Immediately close a socket

gen_udp

The following functions are exported:

- `open(Port) -> {ok, Socket} | {error, Reason}`
  [page ??] Associate a UDP port number with the process calling it
- `open(Port, Options) -> {ok, Socket} | {error, Reason}`
  [page ??] Associate a UDP port number with the process calling it
- `send(Socket, Address, Port, Packet) -> ok | {error, Reason}`
  [page ??] Send a packet
- `recv(Socket, Length) -> {ok, {Address, Port, Packet}} | {error, Reason}`
  [page ??] Receive a packet from a passive socket
- `recv(Socket, Length, Timeout) -> {ok, {Address, Port, Packet}} | {error, Reason}`
  [page ??] Receive a packet from a passive socket
- `controlling_process(Socket, Pid) -> ok`
  [page ??] Change controlling process of a socket
- `close(Socket) -> ok | {error, Reason}`
  [page ??] Close a UDP socket

global

The following functions are exported:

- `del_lock(Id)`
  [page ??] Delete a lock
- `del_lock(Id, Nodes) -> void()`
  [page ??] Delete a lock
- `notify_all_name(Name, Pid1, Pid2) -> none`
  [page ??] Name resolving function that notifies both pids
- `random_exit_name(Name, Pid1, Pid2) -> Pid1 | Pid2`
  [page ??] Name resolving function that kills one pid
- `random_notify_name(Name, Pid1, Pid2) -> Pid1 | Pid2`
  [page ??] Name resolving function that notifies one pid
- `register_name(Name, Pid)`
  [page ??] Globally register a name for a pid
- `register_name(Name, Pid, Resolve) -> yes | no`
  [page ??] Globally register a name for a pid
- `registered_names() -> [Name]`
  [page ??] All globally registered names
- `re_register_name(Name, Pid)`
  [page ??] Atomically re-register a name
- `re_register_name(Name, Pid, Resolve) -> void()`
  [page ??] Atomically re-register a name
- `send(Name, Msg) -> Pid`
  [page ??] Send a message to a globally registered pid
- `set_lock(Id)`
  [page ??] Set a lock on the specified nodes
- `set_lock(Id, Nodes)`
  [page ??] Set a lock on the specified nodes
- `set_lock(Id, Nodes, Retries) -> boolean()`
  [page ??] Set a lock on the specified nodes
- `sync() -> void()`
  [page ??] Synchronize the global name server
- `trans(Id, Fun)`
  [page ??] Micro transaction facility
- `trans(Id, Fun, Nodes)`
  [page ??] Micro transaction facility
- `trans(Id, Fun, Nodes, Retries) -> Res | aborted`
  [page ??] Micro transaction facility
- `unregister_name(Name) -> void()`
  [page ??] Remove a globally registered name for a pid
- `whereis_name(Name) -> pid() | undefined`
  [page ??] Get the pid with a given globally registered name

global_group

The following functions are exported:

- `global_groups() -> {GroupName, GroupNames} | undefined`
  [page ??] Return the global group names
- `info() -> [{Item, Info}]`
  [page ??] Information about global groups
- `monitor_nodes(Flag) -> ok`
  [page ??] Subscribe to node status changes
- `own_nodes() -> Nodes`
  [page ??] Return the group nodes
- `registered_names(Where) -> Names`
  [page ??] Return globally registered names
- `send(Name, Msg) -> pid() | {badarg, {Name, Msg}}`
  [page ??] Send a message to a globally registered pid
- `send(Where, Name, Msg) -> pid() | {badarg, {Name, Msg}}`
  [page ??] Send a message to a globally registered pid
- `sync() -> ok`
  [page ??] Synchronize the group nodes
- `whereis_name(Name) -> pid() | undefined`
  [page ??] Get the pid with a given globally registered name
- `whereis_name(Where, Name) -> pid() | undefined`
  [page ??] Get the pid with a given globally registered name

heart

The following functions are exported:

- `set_cmd(Cmd) -> ok | {error, {bad_cmd, Cmd}}`
  [page ??] Set a temporary reboot command
- `clear_cmd() -> ok`
  [page ??] Clear the temporary boot command
- `get_cmd() -> {ok, Cmd}`
  [page ??] Get the temporary reboot command

inet

The following functions are exported:

- `close(Socket) -> ok`
  [page ??] Close a socket of any type
- `get_rc() -> [{Par, Val}]`
  [page ??] Return a list of IP configuration parameters
- `format_error(Posix) -> string()`
  [page ??] Return a descriptive string for an error reason
- `getaddr(Host, Family) -> {ok, Address} | {error, posix()}`
  [page ??] Return the IP-adress for a host

- `getaddrs(Host, Family) -> {ok, Addresses} | {error, posix()}`
  [page ??] Return the IP-adresses for a host
- `gethostbyaddr(Address) -> {ok, Hostent} | {error, posix()}`
  [page ??] Return a hostent record for the host with the given address
- `gethostbyname(Name) -> {ok, Hostent} | {error, posix()}`
  [page ??] Return a hostent record for the host with the given name
- `gethostbyname(Name, Family) -> {ok, Hostent} | {error, posix()}`
  [page ??] Return a hostent record for the host with the given name
- `gethostname() -> {ok, Hostname} | {error, posix()}`
  [page ??] Return the local hostname
- `getopts(Socket, Options) -> OptionValues | {error, posix()}`
  [page ??] Get one or more options for a socket
- `peername(Socket) -> {ok, {Address, Port}} | {error, posix()}`
  [page ??] Return the address and port for the other end of a connection
- `port(Socket) -> {ok, Port}`
  [page ??] Return the local port number for a socket
- `sockname(Socket) -> {ok, {Address, Port}} | {error, posix()}`
  [page ??] Return the local address and port number for a socket
- `setopts(Socket, Options) -> ok | {error, posix()}`
  [page ??] Set one or more options for a socket

init

The following functions are exported:

- `boot(BootArgs) -> void()`
  [page ??] Start the Erlang runtime system
- `get_args() -> [Arg]`
  [page ??] Get all non-flag command line arguments
- `get_argument(Flag) -> {ok, Arg} | error`
  [page ??] Get the values associated with a command line user flag
- `get_arguments() -> Flags`
  [page ??] Get all command line user flags
- `get_plain_arguments() -> [Arg]`
  [page ??] Get all non-flag command line arguments
- `get_status() -> {InternalStatus, ProvidedStatus}`
  [page ??] Get system status information
- `reboot() -> void()`
  [page ??] Take down an Erlang node smoothly
- `restart() -> void()`
  [page ??] Restart the running Erlang node
- `script_id() -> Id`
  [page ??] Get the identity of the used boot script
- `stop() -> void()`
  [page ??] Take down an Erlang node smoothly<

net_adm

The following functions are exported:

- `dns_hostname(Host) -> {ok, Name} | {error, Host}`
  [page ??] Official name of a host
- `host_file() -> Hosts | {error, Reason}`
  [page ??] Read the `.hosts.erlang` file
- `localhost() -> Name`
  [page ??] Name of the local host
- `names() -> {ok, [{Name, Port}]} | {error, Reason}`
  [page ??] Names of Erlang nodes at a host
- `names(Host) -> {ok, [{Name, Port}]} | {error, Reason}`
  [page ??] Names of Erlang nodes at a host
- `ping(Node) -> pong | pang`
  [page ??] Set up a connection to a node
- `world() -> [node()]`
  [page ??] Lookup and connect to all nodes at all hosts in `.hosts.erlang`
- `world(Arg) -> [node()]`
  [page ??] Lookup and connect to all nodes at all hosts in `.hosts.erlang`
- `world_list(Hosts) -> [node()]`
  [page ??] Lookup and connect to all nodes at specified hosts
- `world_list(Hosts, Arg) -> [node()]`
  [page ??] Lookup and connect to all nodes at specified hosts

net_kernel

The following functions are exported:

- `allow(Nodes) -> ok | error`
  [page ??] Limit access to a specified set of nodes
- `connect_node(Node) -> true | false | ignored`
  [page ??] Establish a connection to a node
- `monitor_nodes(Flag) -> ok | Error`
  [page ??] Subscribe to node status change messages
- `monitor_nodes(Flag, Options) -> ok | Error`
  [page ??] Subscribe to node status change messages
- `get_net_ticktime() -> Res`
  [page ??] Get `net_ticktime`
- `set_net_ticktime(NetTicktime) -> Res`
  [page ??] Set `net_ticktime`
- `set_net_ticktime(NetTicktime, TransitionPeriod) -> Res`
  [page ??] Set `net_ticktime`
- `start([Name]) -> {ok, pid()} | {error, Reason}`
  [page ??] Turn an Erlang runtime system into a distributed node
- `start([Name, NameType]) -> {ok, pid()} | {error, Reason}`
  [page ??] Turn an Erlang runtime system into a distributed node

- `start([Name, NameType, Ticktime]) -> {ok, pid()} | {error, Reason}`
  [page ??] Turn an Erlang runtime system into a distributed node
- `stop() -> ok | {error, not_allowed | not_found}`
  [page ??] Turn a node into a non-distributed Erlang runtime system

os

The following functions are exported:

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

packages

The following functions are exported:

- `no functions exported`
  [page ??] x

pg2

The following functions are exported:

- `create(Name) -> void()`
  [page ??] Create a new, empty process group
- `delete(Name) -> void()`
  [page ??] Delete a process group
- `get_closest_pid(Name) -> Pid | {error, Reason}`
  [page ??] Common dispatch function
- `get_members(Name) -> [Pid] | {error, Reason}`
  [page ??] Return all processes in a group

- `get_local_members(Name) -> [Pid] | {error, Reason}`
  [page ??] Return all local processes in a group
- `join(Name, Pid) -> ok | {error, Reason}`
  [page ??] Join a process to a group
- `leave(Name, Pid) -> ok | {error, Reason}`
  [page ??] Make a process leave a group
- `which_groups() -> [Name]`
  [page ??] Return a list of all known groups
- `start()`
  [page ??] Start the pg2 server
- `start_link() -> {ok, Pid} | {error, Reason}`
  [page ??] Start the pg2 server

rpc

The following functions are exported:

- `call(Node, Module, Function, Args) -> Res | {badrpc, Reason}`
  [page ??] Evaluate a function call on a node
- `call(Node, Module, Function, Args, Timeout) -> Res | {badrpc, Reason}`
  [page ??] Evaluate a function call on a node
- `block_call(Node, Module, Function, Args) -> Res | {badrpc, Reason}`
  [page ??] Evaluate a function call on a node in the RPC server's context
- `block_call(Node, Module, Function, Args, Timeout) -> Res | {badrpc, Reason}`
  [page ??] Evaluate a function call on a node in the RPC server's context
- `async_call(Node, Module, Function, Args) -> Key`
  [page ??] Evaluate a function call on a node, asynchrous version
- `yield(Key) -> Res | {badrpc, Reason}`
  [page ??] Deliver the result of evaluating a function call on a node (blocking)
- `nb_yield(Key) -> {value, Val} | timeout`
  [page ??] Deliver the result of evaluating a function call on a node (non-blocking)
- `nb_yield(Key, Timeout) -> {value, Val} | timeout`
  [page ??] Deliver the result of evaluating a function call on a node (non-blocking)
- `multicall(Module, Function, Args) -> {ResL, BadNodes}`
  [page ??] Evaluate a function call on a number of nodes
- `multicall(Nodes, Module, Function, Args) -> {ResL, BadNodes}`
  [page ??] Evaluate a function call on a number of nodes
- `multicall(Module, Function, Args, Timeout) -> {ResL, BadNodes}`
  [page ??] Evaluate a function call on a number of nodes
- `multicall(Nodes, Module, Function, Args, Timeout) -> {ResL, BadNodes}`
  [page ??] Evaluate a function call on a number of nodes
- `cast(Node, Module, Function, Args) -> void()`
  [page ??] Run a function on a node ignoring the result

- `eval_everywhere(Module, Funtion, Args) -> void()`
  [page ??] Run a function on all nodes, ignoring the result
- `eval_everywhere(Nodes, Module, Function, Args) -> void()`
  [page ??] Run a function on specific nodes, ignoring the result
- `abcast(Name, Msg) -> void()`
  [page ??] Broadcast a message asynchronously to a registered process on all nodes
- `abcast(Nodes, Name, Msg) -> void()`
  [page ??] Broadcast a message asynchronously to a registered process on specific nodes
- `sbcast(Name, Msg) -> {GoodNodes, BadNodes}`
  [page ??] Broadcast a message synchronously to a registered process on all nodes
- `sbcast(Nodes, Name, Msg) -> {GoodNodes, BadNodes}`
  [page ??] Broadcast a message synchronously to a registered process on specific nodes
- `server_call(Node, Name, ReplyWrapper, Msg) -> Reply | {error, Reason}`
  [page ??] Interact with a server on a node
- `multi_server_call(Name, Msg) -> {Replies, BadNodes}`
  [page ??] Interact with the servers on a number of nodes
- `multi_server_call(Nodes, Name, Msg) -> {Replies, BadNodes}`
  [page ??] Interact with the servers on a number of nodes
- `safe_multi_server_call(Name, Msg) -> {Replies, BadNodes}`
  [page ??] Interact with the servers on a number of nodes (deprecated)
- `safe_multi_server_call(Nodes, Name, Msg) -> {Replies, BadNodes}`
  [page ??] Interact with the servers on a number of nodes (deprecated)
- `parallel_eval(FuncCalls) -> ResL`
  [page ??] Evaluate several function calls on all nodes in parallel
- `pmap({Module, Function}, ExtraArgs, List2) -> List1`
  [page ??] Parallell evaluation of mapping a function over a list
- `pinfo(Pid) -> [{Item, Info}] | undefined`
  [page ??] Information about a process
- `pinfo(Pid, Item) -> {Item, Info} | undefined | []`
  [page ??] Information about a process

seq_trace

The following functions are exported:

- `set_token(Token) -> PreviousToken`
  [page ??] Set the trace token
- `set_token(Component, Val) -> {Component, OldVal}`
  [page ??] Set a component of the trace token
- `get_token() -> TraceToken`
  [page ??] Return the value of the trace token
- `get_token(Component) -> {Component, Val}`
  [page ??] Return the value of a trace token component

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

user

No functions are exported.

wrap_log_reader

The following functions are exported:

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

zlib

The following functions are exported:

- `open() -> Z`
  [page ??] Open a stream and return a stream reference
- `close(Z) -> ok`
  [page ??] Close a stream
- `deflateInit(Z) -> ok`
  [page ??] Initialize a session for compression
- `deflateInit(Z, Level) -> ok`
  [page ??] Initialize a session for compression
- `deflateInit(Z, Level, Method, WindowBits, MemLevel, Strategy) -> ok`
  [page ??] Initialize a session for compression
- `deflate(Z, Data) -> Compressed`
  [page ??] Compress data

- `deflate(Z, Data, Flush) ->`
  [page ??] Compress data
- `deflateSetDictionary(Z, Dictionary) -> Adler32`
  [page ??] Initialize the compression dictionary
- `deflateReset(Z) -> ok`
  [page ??] Reset the deflate session
- `deflateParams(Z, Level, Strategy) -> ok`
  [page ??] Dynamicly update deflate parameters
- `deflateEnd(Z) -> ok`
  [page ??] End deflate session
- `inflateInit(Z) -> ok`
  [page ??] Initialize a session for decompression
- `inflateInit(Z, WindowBits) -> ok`
  [page ??] Initialize a session for decompression
- `inflate(Z, Data) -> DeCompressed`
  [page ??] Decompress data
- `inflateSetDictionary(Z, Dictionary) -> ok`
  [page ??] Initialize the decompression dictionary
- `inflateReset(Z) -> ok`
  [page ??] >Reset the inflate session
- `inflateEnd(Z) -> ok`
  [page ??] End inflate session
- `setBufSize(Z, Size) -> ok`
  [page ??] Set buffer size
- `getBufSize(Z) -> Size`
  [page ??] Get buffer size
- `crc32(Z) -> CRC`
  [page ??] Get current CRC
- `crc32(Z, Binary) -> CRC`
  [page ??] Calculate CRC
- `crc32(Z, PrevCRC, Binary) -> CRC`
  [page ??] Calculate CRC
- `adler32(Z, Binary) -> Checksum`
  [page ??] Calculate the adler checksum
- `adler32(Z, PrevAdler, Binary) -> Checksum`
  [page ??] Calculate the adler checksum
- `compress(Binary) -> Compressed`
  [page ??] Compress a binary with standard zlib functionality
- `uncompress(Binary) -> Decompressed`
  [page ??] Uncompress a binary with standard zlib functionality
- `zip(Binary) -> Compressed`
  [page ??] Compress a binary without the zlib headers
- `unzip(Binary) -> Decompressed`
  [page ??] Uncompress a binary without the zlib headers
- `gzip(Data) -> Compressed`
  [page ??] Compress a binary with gz header
- `gunzip(Bin) -> Decompressed`
  [page ??] Uncompress a binary with gz header

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 standard error logger event handlers are defined in the Kernel application. These are described in error_logger(3) [page ??].

Configuration

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

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

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

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

`distributed = [Distrib]` Specifies which applications are distributed and on which nodes they may execute. In this parameter:

- `Distrib = {App,Nodes} | {App,Time,Nodes}`
- `App = atom()`
- `Time = integer()>0`
- `Nodes = [node() | {node(),...,node()}]`

The parameter is described in `application(3)`, function `load/2`.

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

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

`once` Connections will be established automatically, but only once per node. If a node goes down, it must thereafter be explicitly connected. See `net_kernel(3)`.

`permissions = [Perm]` Specifies the default permission for applications when they are started. In this parameter:

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

Permissions are described in `application(3)`, function `permit/2`.

`error_logger = Value` `Value` is one of:

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

`{file, FileName}` Installs the standard event handler which prints error reports to the file `FileName`, where `FileName` is a string.

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

`silent` Error logging is turned off.

`global_groups = [GroupTuple]` Defines global groups, see `global_group(3)`.

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

- PublishType = normal | hidden
- Node = node()

inet_default_connect_options = [{Opt, Val}] Specifies default options for connect sockets, see inet(3).

inet_default_listen_options = [{Opt, Val}] Specifies default options for listen (and accept) sockets, see inet(3).

{inet_dist_use_interface, ip_address()} If the host of an Erlang node has several network interfaces, this parameter specifies which one to listen on. See inet(3) for the type definition of ip_address().

{inet_dist_listen_min, First} See below.

{inet_dist_listen_max, Last} Define the First..Last port range for the listener socket of a distributed Erlang node.

inet_parse_error_log = silent If this configuration parameter is set, no error_logger messages are generated when erroneous lines are found and skipped in the various Inet configuration files.

inetrc = Filename The name (string) of an Inet user configuration file. See ERTS User's Guide, Inet configuration.

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

Note that this value does not limit the total connection setup time, but rather each individual network operation during the connection setup and handshake.

net_ticktime = TickTime Specifies the net_kernel tick time. TickTime is given in seconds. Once every TickTime/4 second, all connected nodes are ticked (if anything else has been written to a node) and if nothing has been received from another node within the last four (4) tick times that node is considered to be down. This ensures that nodes which are not responding, for reasons such as hardware errors, are considered to be down.

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

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

TickTime is by default 60 (seconds). Thus, 45 < T < 75 seconds.

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

Note: Normally, a terminating node is detected immediately.

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

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

sync_nodes_timeout = integer() | infinity Specifies the amount of time (in milliseconds) this node will wait for the mandatory and optional nodes to start. If this parameter is undefined, no node synchronization is performed. This option also makes sure that global is synchronized.

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

The default value is `false`. If this parameter is undefined, the server is started if the parameter `distributed` is set.

`start_boot_server = true | false` Starts the `boot_server` if the parameter is `true` (see `erl_boot_server(3)`). This parameter should be set to `true` in an embedded system which uses this service.

The default value is `false`.

`boot_server_slaves = [SlaveIP]` If the `start_boot_server` configuration parameter is `true`, this parameter can be used to initialize `boot_server` with a list of slave IP addresses. `SlaveIP = string() | atom | {integer(),integer(),integer(),integer()}`

where `0 <= integer() <=255`.

Examples of `SlaveIP` in atom, string and tuple form are: `'150.236.16.70', "150,236,16,70", {150,236,16,70}`.

The default value is `[]`.

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

The default value is `false`.

`start_pg2 = true | false` Starts the pg2 server (see `pg2(3)`) if the parameter is `true`. This parameter should be set to `true` in an embedded system which uses this service.

The default value is `false`.

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

The default value is `false`.

`shutdown_func = {Mod, Func}` Where:

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

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

See Also

app(4) [page ??], application(3) [page ??], code(3) [page ??], disk_log(3) [page ??], erl_boot_server(3) [page ??], erl_ddll(3) [page ??], error_logger(3) [page ??], file(3) [page ??], global(3) [page ??], global_group(3) [page ??], heart(3) [page ??], inet(3) [page ??], net_kernel(3) [page ??], os(3) [page ??], pg2(3) [page ??], rpc(3) [page ??], seq_trace(3) [page ??], user(3) [page ??]

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) [page ??] 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
set_env(Application, Par, Val, Timeout) -> ok
```

Types:

- Application = atom()
- Par = atom()
- Val = term()
- Timeout = int() | infinity

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

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

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

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

Types:

- Application = atom()
- Type = 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
unset_env(Application, Par, Timeout) -> ok
```

Types:

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

- Timeout = int() | infinity

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

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

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

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

Types:

- Application = atom()
- Description = string()
- Vsn = string()
- Timeout = int() | infinity

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.

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

CALLBACK MODULE

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

Exports

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

Types:

- StartType = normal | {takeover,Node} | {failover,Node}
- Node = node()
- StartArgs = term()
- Pid = pid()
- State = term()

This function is called whenever an application is started using `application:start/1,2`, and should start the processes of the application. If the application is structured according to the OTP design principles as a supervision tree, this means starting the top supervisor of the tree.

`StartType` defines the type of start:

- `normal` if 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 ??], app(4) [page ??]

auth

Erlang Module

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

Exports

`is_auth(Node) -> yes | no`

Types:

- Node = node()

Returns `yes` if communication with `Node` is authorized. Note that a connection to `Node` will be established in this case. Returns `no` if `Node` does not exist or communication is not authorized (it has another cookie than `auth` thinks it has).

Use net_adm:ping(Node) [page ??] instead.

`cookie() -> Cookie`

Types:

- Cookie = atom()

Use erlang:get_cookie() [page ??] instead.

`cookie(TheCookie) -> true`

Types:

- TheCookie = Cookie | [Cookie]
  The cookie may also be given as a list with a single atom element
- Cookie = atom()

Use erlang:set_cookie(node(), Cookie) [page ??] instead.

`node_cookie([Node, Cookie]) -> yes | no`

Types:

- Node = node()
- Cookie = atom()

Equivalent to node_cookie(Node, Cookie) [page ??].

`node_cookie(Node, Cookie) -> yes | no`

Types:

- Node = node()

- Cookie = atom()

Sets the magic cookie of `Node` to `Cookie`, and verifies the status of the authorization. Equivalent to calling erlang:set_cookie(Node, Cookie) [page ??], followed by auth:is_auth(Node) [page ??].

code

Erlang Module

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

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

```
% erl -mode interactive
```

Default mode is `interactive`.

- In embedded mode, all code is loaded during system start-up according to the boot script. (Code can also be loaded later by explicitly ordering the code server to do so).
- In interactive mode, only some code is loaded during system startup-up, basically the modules needed by the runtime system itself. Other code is dynamically loaded when first referenced. When a call to a function in a certain module is made, and the module is not loaded, the code server searches for and tries to load the module.

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

Code Path

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

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

Code Path Cache

The code server incorporates a code path cache. The cache functionality is disabled by default. To activate it, start the emulator with the command line flag `-code_path_cache` or call `code:rehash()`. When the cache is created (or updated), the code server searches for modules in the code path directories. This may take some time if the the code path is long. After the cache creation, the time for loading modules in a large system (one with a large directory structure) is significantly reduced compared to having the cache disabled. The code server is able to look up the location of a module from the cache in constant time instead of having to search through the code path directories.

Application resource files (`.app` files) are also stored in the code path cache. This feature is used by the application controller (see application(3) [page ??]) 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.

Current and Old Code

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

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

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

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

For more information about old and current code, and how to make a process switch from old to current code, refer to [Erlang Reference Manual].

Exports

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

Types:

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

Sets the code path to the list of directories Path.

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

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

Types:

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

Returns the code path

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

Types:

- Dir = string()
- What = bad_directory

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

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

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

Types:

- Dir = string()
- What = bad_directory

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

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

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

Types:

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

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

```
add_pathsa(Dirs) -> ok
```

Types:

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

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

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

Types:

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

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

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

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

Types:

- Name = atom()
- Dir = string()
- What = bad_name | bad_directory | {badarg, term()}

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

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

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

Types:

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

Tries to load the Erlang module Module, using the code path. It looks for the object code file with an extension that corresponds to the Erlang machine used, for example Module.beam. The loading fails if the module name found in the object code differs from the name Module. load_binary/3 [page ??] must be used to load object code with a module name that is different from the file name.

Returns {module, Module} if successful, or {error, nofile} if no object code is found, or {error, sticky_directory} if the object code resides in a sticky directory, or {error, badarg} if the argument is invalid. Also if the loading fails, an error tuple is returned. See erlang:load_module/2 [page ??] for possible values of What.

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

Types:

- Filename = string()

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

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

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

Types:

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

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

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

Types:

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

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

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

`delete(Module) -> true | false`

Types:

- Module = atom()

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

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

`purge(Module) -> true | false`

Types:

- Module = atom()

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

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

soft_purge(Module) -> true | false

Types:

- Module = atom()

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

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

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

Types:

- Module = atom()
- Loaded = Absname | preloaded | cover_compiled
- Absname = string()

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

Normally, Loaded is the absolute file name Absname from which the code was obtained. If the module is preloaded (see [script(4)]), Loaded==preloaded. If the module is Cover compiled (see [cover(3)]), Loaded==cover_compiled.

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

Types:

- Module = atom()
- Loaded = Absname | preloaded | cover_compiled
- Absname = string()

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

which(Module) -> Which

Types:

- Module = atom()
- Which = Filename | non_existing | preloaded | cover_compiled
- Filename = string()

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

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

Types:

- Module = atom()

- Binary = binary()
- Filename = string()

Searches the code path for the object code of the module `Module`. It returns {`Module, Binary, Filename`} if successful, and `error` if not. `Binary` is a binary data object which contains the object code for the module. This can be useful if code is to be loaded on a remote node in a distributed system. For example, loading module `Module` on a node `Node` is done as follows:

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

`root_dir() -> string()`

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

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

`lib_dir() -> string()`

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

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

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

Types:

- Name = atom()

This function is mainly intended for finding out the path for the "library directory", the top directory, for an application `Name` located under `$OTPROOT/lib`.

If there is a directory called `Name` in the code path, optionally with a `-Vsn` suffix and/or an `ebin` subdirectory, the name of this directory is returned.

```
> code:lib_dir(mnesia).
"/usr/local/otp/lib/mnesia-4.2.2"
```

Returns {`error, bad_name`} if `Name` is not found or if the argument is not valid.

`compiler_dir() -> string()`

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

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

Types:

- Name = atom()

This function is mainly intended for finding out the path for the priv directory for an application Name located under $OTPROOT/lib.

If there is a directory called Name in the code path, optionally with a -Vsn suffix and/or an ebin subdirectory, the function returns the name of this directory with priv appended. It is not checked if this directory really exists.

```
> code:priv_dir(mnesia).
"/usr/local/otp/lib/mnesia-4.2.2/priv"
```

Returns {error, bad_name} if Name is not found or if the argument is not valid.

`objfile_extension() -> ".beam"`

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

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

Types:

- Dir = string()
- What = term()

This function marks Dir as sticky.

Returns ok if successful, and an error tuple otherwise.

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

Types:

- Dir = string()
- What = term()

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

Returns ok if successful, and an error tuple otherwise.

`rehash() -> ok`

This function creates or rehashes the code path cache.

`where_is_file(Filename) -> Absname | non_existing`

Types:

- Filename = Absname = string()

Searches the code path for Filename, a file of arbitrary type. If found, the full name is returned. non_existing is returned if the file cannot be found. The function can be useful, for example, to locate application resource files. If the code path cache is used, the code server will efficiently read the full name from the cache, provided that Filename is an object code file or an .app file.

`clash() -> ok`

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

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 ??]. There can be several owners of a log, but a process cannot own a log more than once. One and the same process may, however, open the log as a user more than once. For a disk log process to properly close its file and terminate, it must be closed by its owners and once by some non-owner process for each time the log was used anonymously; the users are counted, and there must not be any users left when the disk log process terminates.

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

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

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

A collection of open disk logs with the same name running on different nodes is said to be a *a distributed disk log* if requests made to any one of the logs are automatically made to the other logs as well. The members of such a collection will be called individual distributed disk logs, or just distributed disk logs if there is no risk of confusion. There is no order between the members of such a collection. For instance, logged terms are not necessarily written onto the node where the request was made before written onto the other nodes. One could note here that there are a few functions that do not make requests to all members of distributed disk logs, namely `info`, `chunk`, `bchunk`, `chunk_step` and `lclose`. An open disk log that is not a distributed disk log is said to be a *local disk log*. A local disk log is accessible only from the node where the disk log process runs, whereas a distributed disk log is accessible from all nodes in the Erlang system, with exception for those nodes where a local disk log with the same name as the distributed disk log exists. All processes on nodes that have access to a local or distributed disk log can log items or otherwise change, inspect or close the log.

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

> **Note:**
> In some applications it may not be acceptable that replies from individual logs are ignored. An alternative in such situations is to use several local disk logs instead of one distributed disk log, and implement the distribution without use of the disk log module.

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

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

The error message `no_such_log` means that the given disk log is not currently open. Nothing is said about whether the disk log files exist or not.

> **Note:**
> If an attempt to reopen or truncate a log fails (see `reopen` and `truncate`) the disk log process immediately terminates. Before the process terminates links to to owners and blocking processes (see `block`) are removed. The effect is that the links work in one direction only; any process using a disk log has to check for the error message `no_such_log` if some other process might truncate or reopen the log simultaneously.

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) -> Chars`

Types:

- Chars = [char() | Chars]

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 ??].
- {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 ??].

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 ??], pg2(3) [page ??], wrap_log_reader(3) [page ??]

erl_boot_server

Erlang Module

This server is used to assist diskless Erlang nodes which fetch all Erlang code from another machine.

This server is used to fetch all code, including the start script, if an Erlang runtime system is started with the `-loader inet` command line flag. All hosts specified with the `-hosts Host` command line flag must have one instance of this server running.

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

Exports

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

Types:

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

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 = term()()

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 = term()

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 ??], erl_prim_loader(3) [page ??]

erl_ddll

Erlang Module

The `erl_ddll` module provides an interface for loading and unloading *erlang linked in drivers* in runtime.

> **Note:**
> This is a large reference document. For casual use of the module, as well as for most real world applications, the descriptions of the functions load/2 [page ??] and unload/1 [page ??] are enough to get going.

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

When describing a set of functions, (i.e. a module, a part of a module or an application) executing in a process and wanting to use a ddll-driver, we use the term *user*. There can be several users in one process (different modules needing the same driver) and several processes running the same code, making up several *users* of a driver. In the basic scenario, each user loads the driver before starting to use it and unloads the driver when done. The reference counting keeps track of processes as well as the number of loads by each process, so that the driver will only be unloaded when no one wants it (it has no user). The driver also keeps track of ports that are opened towards it, so that one can delay unloading until all ports are closed or kill all ports using the driver when it is unloaded.

The interface supports two basic scenarios of loading and unloading. Each scenario can also have the option of either killing ports when the driver is unloading, or waiting for the ports to close themselves. The scenarios are:

Load and unload on a "when needed basis" This (most common) scenario simply supports that each user [page ??] of the driver loads it when it is needed and unloads it when the user [page ??] no longer have any use for it. The driver is always reference counted and as long as a process keeping the driver loaded is still alive, the driver is present in the system.

Each user [page ??] of the driver use *literally* the same pathname for the driver when demanding load, but the users [page ??] are not really concerned with if the driver is already loaded from the filesystem or if the object code has to be loaded from filesystem.

Two pairs of functions support this scenario:

load/2 and unload/1 When using the `load/unload` interfaces, the driver will not *actually* get unloaded until the *last port* using the driver is closed. The function `unload/1` can return immediately, as the users [page ??] are not really concerned with when the actual unloading occurs. The driver will actually get unloaded when no one needs it any longer.
If a process having the driver loaded dies, it will have the same effect as if unloading was done.
When loading, the function `load/2` returns `ok` as soon as there is any instance of the driver present, so that if a driver is waiting to get unloaded (due to open ports), it will simply change state to no longer need unloading.

load_driver/2 and unload_driver/1 These interfaces is intended to be used when it is considered an error that ports are open towards a driver that no user [page ??] has loaded. The ports still open when the last user [page ??] calls `unload_driver/1` or when the last process having the driver loaded dies, will get killed with reason `driver_unloaded`.
The function names `load_driver` and `unload_driver` are kept for backward compatibility.

Loading and reloading for code replacement This scenario occurs when the driver code might need replacement during operation of the Erlang emulator. Implementing driver code replacement is somewhat more tedious than beam code replacement, as one driver cannot be loaded as both “old” and “new” code. All users [page ??] of a driver must have it closed (no open ports) before the old code can be unloaded and the new code can be loaded.

The actual unloading/loading is done as one atomic operation, blocking all processes in the system from using the driver concerned while in progress.

The preferred way to do driver code replacement is to let *one single process* keep track of the driver. When the process start, the driver is loaded. When replacement is required, the driver is reloaded. Unload is probably never done, or done when the process exits. If more than one user [page ??] has a driver loaded when code replacement is demanded, the replacement cannot occur until the last “other” user [page ??] has unloaded the driver.

Demanding reload when a reload is already in progress is always an error. Using the high level functions, it is also an error to demand reloading when more than one user [page ??] has the driver loaded. To simplify driver replacement, avoid designing your system so that more than than one user [page ??] has the driver loaded.

The two functions for reloading drivers should be used together with corresponding load functions, to support the two different behaviors concerning open ports:

load/2 and reload/2 This pair of functions is used when reloading should be done after the last open port towards the driver is closed.
As `reload/2` actually waits for the reloading to occur, a misbehaving process keeping open ports towards the driver (or keeping the driver loaded) might cause infinite waiting for reload. Timeouts has to be provided outside of the process demanding the reload or by using the low-level interface try_load/3 [page ??] in combination with driver monitors (see below).

load_driver/2 and reload_driver/2 This pair of functions are used when open ports towards the driver should be killed with reason `driver_unloaded` to allow for new driver code to get loaded.
If, however, another process has the driver loaded, calling `reload_driver` returns the error code `pending_process`. As stated earlier, the recommended

design is to not allow other users [page ??] than the “driver reloader” to actually demand loading of the concerned driver.

Exports

`demonitor(MonitorRef) -> ok`

Types:

- MonitorRef = ref()

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

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

`info() -> AllInfoList`

Types:

- AllInfoList = [DriverInfo]
- DriverInfo = {DriverName, InfoList}
- DriverName = string()
- InfoList = [InfoItem]
- InfoItem = {Tag, Value}
- Tag = atom()
- Value = term()

Returns a list of tuples {`DriverName, InfoList`}, where `InfoList` is the result of calling info/1 [page ??] for that `DriverName`. Only dynamically linked in drivers are included in the list.

`info(Name) -> InfoList`

Types:

- Name = string() | atom()
- InfoList = [InfoItem]
- InfoItem = {Tag, Value}
- Tag = atom()
- Value = term()

Returns a list of tuples {`Tag, Value`}, where `Tag` is the information item and `Value` is the result of calling info/2 [page ??] with this driver name and this tag. The result being a tuple list containing all information available about a driver.

The different tags that will appear in the list are:

- processes
- driver_options
- port_count
- linked_in_driver
- permanent

- awaiting_load
- awaiting_unload

For a detailed description of each value, please read the description of info/2 [page ??] below.

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

`info(Name, Tag) -> Value`

Types:

- Name = string() | atom()
- Tag = processes | driver_options | port_count | linked_in_driver | permanent | awaiting_load | awaiting_unload
- Value = term()

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

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

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

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

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

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

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

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

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

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

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

Types:

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

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

The `Name` given should correspond to the filename of the actual dynamically loadable object file residing in the directory given as `Path`, but *without* the extension (i.e. `.so`). The driver name provided in the driver initialization routine must correspond with the filename, in much the same way as erlang module names correspond to the names of the `.beam` files.

If the driver has been previously unloaded, but is still present due to open ports against it, a call to `load/2` will stop the unloading and keep the driver (as long as the `Path` is the same) and `ok` is returned. If one actually wants the object code to be reloaded, one uses reload/2 [page ??] or the low-level interface try_load/3 [page ??] instead. Please refer to the description of different scenarios [page ??] for loading/unloading in the introduction.

If more than one process tries to load an already loaded driver withe the same `Path`, or if the same process tries to load it several times, the function will return `ok`. The emulator will keep track of the `load/2` calls, so that a corresponding number of `unload/2` calls will have to be done from the same process before the driver will actually get unloaded. It is therefore safe for an application to load a driver that is shared between processes or applications when needed. It can safely be unloaded without causing trouble for other parts of the system.

It is not allowed to load several drivers with the same name but with different `Path` parameters.

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

On success, the function returns `ok`. On failure, the return value is {`error,ErrorDesc`}, where `ErrorDesc` is an opaque term to be translated into human readable form by the format_error/1 [page ??] function.

For more control over the error handling, again use the try_load/3 [page ??] interface instead.

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

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

Types:

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

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

The number of loads and unloads by different users [page ??] influence the actual loading and unloading of a driver file. The port killing will therefore only happen when the *last* user [page ??] unloads the driver, or the last process having loaded the driver exits.

This interface (or at least the name of the functions) is kept for backward compatibility. Using try_load/3 [page ??] with `{driver_options,[kill_ports]}` in the option list will give the same effect regarding the port killing.

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

`monitor(Tag, Item) -> MonitorRef`

Types:

- Tag = driver
- Item = {Name, When}
- Name = atom() | string()
- When = loaded | unloaded | unloaded_only
- MonitorRef = ref()

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

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

The `MonitorRef` can also be used in subsequent calls to demonitor/1 [page ??] to remove a monitor.

The function accepts the following parameters:

Tag The monitor tag is always `driver` as this function can only be used to create driver monitors. In the future, driver monitors will be integrated with process monitors, why this parameter has to be given for consistence.

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

loaded Notify me when the driver is reloaded (or loaded if loading is underway). It only makes sense to monitor drivers that are in the process of being loaded or reloaded. One cannot monitor a future-to-be driver name for loading, that will only result in a `'DOWN'` message being immediately sent. Monitoring for loading is therefore most useful when triggered by the try_load/3 [page ??] function, where the monitor is created *because* the driver is in such a pending state.
Setting a driver monitor for `loading` will eventually lead to one of the following messages being sent:

{***'UP', ref(), driver, Name, loaded***} This message is sent, either immediately if the driver is already loaded and no reloading is pending, or when reloading is executed if reloading is pending.
The user [page ??] is expected to know if reloading is demanded prior to creating a monitor for loading.

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

{***'DOWN', ref(), driver, Name, load_cancelled***} This message will arrive if reloading was underway, but the user [page ??] having requested reload cancelled it by either dying or calling try_unload/2 [page ??] (or `unload/1`/`unload_driver/1`) again before it was reloaded.

{***'DOWN', ref(), driver, Name,*** {***load_failure, Failure***}} This message will arrive if reloading was underway but the loading for some reason failed. The `Failure` term is one of the errors that can be returned from try_load/3 [page ??]. The error term can be passed to format_error/1 [page ??] for translation into human readable form. Note that the translation has to be done in the same running erlang virtual machine as the error was detected in.

unloaded Monitor when a driver gets unloaded. If one monitors a driver that is not present in the system, one will immediately get notified that the driver got unloaded. There is no guarantee that the driver was actually ever loaded.
A driver monitor for unload will eventually result in one of the following messages being sent:

{***'DOWN', ref(), driver, Name, unloaded***} The driver instance monitored is now unloaded. As the unload might have been due to a `reload/2` request, the driver might once again have been loaded when this message arrives.

{***'UP', ref(), driver, Name, unload_cancelled***} This message will be sent if unloading was expected, but while the driver was waiting for all ports to get closed, a new user [page ??] of the driver appeared and the unloading was cancelled.
This message appears when an {`ok, pending_driver`}) was returned from try_unload/2 [page ??]) for the last user [page ??] of the driver and then a {`ok, already_loaded`} is returned from a call to try_load/3 [page ??].
If one wants to *really* monitor when the driver gets unloaded, this message will distort the picture, no unloading was really done. The `unloaded_only` option creates a monitor similar to an `unloaded` monitor, but does never result in this message.

{***'UP', ref(), driver, Name, permanent***} This message will be sent if unloading was expected, but the driver made itself permanent prior to unloading. It will also be sent if trying to monitor a permanent or statically linked in driver.

unloaded_only A monitor created as `unloaded_only` behaves exactly as one created as `unloaded` with the exception that the {`'UP', ref(), driver, Name, unload_cancelled`} message will never be sent, but the monitor instead persists until the driver *really* gets unloaded.

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

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

Types:

- Path = Name = string() | atom()
- ErrorDesc = pending_process | OpaqueError
- OpaqueError = term()

Reloads the driver named `Name` from a possibly different `Path` than was previously used. This function is used in the code change scenario [page ??] described in the introduction.

If there are other users [page ??] of this driver, the function will return {`error, pending_process`}, but if there are no more users, the function call will hang until all open ports are closed.

> **Note:**
> Avoid mixing several users [page ??] with driver reload requests.

If one wants to avoid hanging on open ports, one should use the try_load/3 [page ??] function instead.

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

> **Note:**
> Avoid mixing several users [page ??] with driver reload requests.

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

For more control over the error handling, again use the try_load/3 [page ??] interface instead.

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

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

Types:

- Path = Name = string() | atom()
- ErrorDesc = pending_process | OpaqueError
- OpaqueError = term()

Works exactly as reload/2 [page ??], but for drivers loaded with the load_driver/2 [page ??] interface.

As this interface implies that ports are being killed when the last user disappears, the function wont hang waiting for ports to get closed.

For further details, see the scenarios [page ??] in the module description and refer to the reload/2 [page ??] function description.

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

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

Types:

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

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

{***ok, loaded***} The driver was actually loaded and is immediately usable.

{***ok, already_loaded***} The driver was already loaded by another process and/or is in use by a living port. The load by you is registered and a corresponding `try_unload` is expected sometime in the future.

{***ok, pending_driver***} or {***ok, pending_driver, ref()***} The load request is registered, but the loading is delayed due to the fact that an earlier instance of the driver is still waiting to get unloaded (there are open ports using it). Still, unload is expected when you are done with the driver. This return value will *mostly* happen when the {`reload,pending_driver`} or {`reload,pending`} options are used, but *can* happen when another user [page ??] is unloading a driver in parallel and the `kill_ports` driver option is set. In other words, this return value will always need to be handled!

{***ok, pending_process***} or {***ok, pending_process, ref()***} The load request is registered, but the loading is delayed due to the fact that an earlier instance of the driver is still waiting to get unloaded by another user [page ??] (not only by a port, in which case {`ok,pending_driver`} would have been returned). Still, unload is expected when you are done with the driver. This return value will *only* happen when the {`reload,pending`} option is used.

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

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

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

The function accepts the following parameters:

Path The filesystem path to the directory where the driver object file is situated. The filename of the object file (minus extension) must correspond to the driver name (used in the name parameter) and the driver must identify itself with the very same name. The Path might be provided as an *io_list*, meaning it can be a list of other io_lists, characters (eight bit integers) or binaries, all to be flattened into a sequence of characters.

The (possibly flattened) Path parameter must be consistent throughout the system, a driver should, by all users [page ??], be loaded using the same *literal* Path. The exception is when *reloading* is requested, in which case the Path may be specified differently. Note that all users [page ??] trying to load the driver at a later time will need to use the *new* Path if the Path is changed using a reload option. This is yet another reason to have *only one loader* of a driver one wants to upgrade in a running system!

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

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

{***driver_options, DriverOptionsList***} This option is to provide options that will change it's general behavior and will "stick" to the driver throughout it's lifespan.

The driver options for a given driver name need always to be consistent, *even when the driver is reloaded*, meaning that they are as much a part of the driver as the actual name.

Currently the only allowed driver option is kill_ports, which means that all ports opened towards the driver are killed with the exit-reason driver_unloaded when no process any longer has the driver loaded. This situation arises either when the last user [page ??] calls try_unload/2 [page ??], or the last process having loaded the driver exits.

{***monitor, MonitorOption***} A `MonitorOption` tells `try_load/3` to trigger a driver monitor under certain conditions. When the monitor is triggered, the function will return a three-tuple {`ok, PendingStatus, ref()`}, where the `ref()` is the monitor ref for the driver monitor.
Only one `MonitorOption` can be specified and it is either the atom `pending`, which means that a monitor should be created whenever a load operation is delayed, and the atom `pending_driver`, in which a monitor is created whenever the operation is delayed due to open ports towards an otherwise unused driver. The `pending_driver` option is of little use, but is present for completeness, it is very well defined which reload-options might give rise to which delays. It might, however, be a good idea to use the same `MonitorOption` as the `RelaodOption` if present.
If reloading is not requested, it might still be useful to specify the `monitor` option, as forced unloads (`kill_ports` driver option or the `kill_ports` option to try_unload/2 [page ??]) will trigger a transient state where driver loading cannot be performed until all closing ports are actually closed. So, as `try_unload` can, in almost all situations, return {`ok, pending_driver`}, one should always specify at least {`monitor, pending_driver`} in production code (see the monitor discussion above).

{***reload,RealoadOption***} This option is used when one wants to *reload* a driver from disk, most often in a code upgrade scenario. Having a `reload` option also implies that the `Path` parameter need *not* be consistent with earlier loads of the driver.
To reload a driver, the process needs to have previously loaded the driver, i.e there has to be an active user [page ??] of the driver in the process.
The `reload` option can be either the atom `pending`, in which reloading is requested for any driver and will be effectuated when *all* ports opened against the driver are closed. The replacement of the driver will in this case take place regardless of if there are still pending users [page ??] having the driver loaded! The option also triggers port-killing (if the `kill_ports` driver option is used) even though there are pending users, making it usable for forced driver replacement, but laying a lot of responsibility on the driver users [page ??]. The pending option is seldom used as one does not want other users [page ??] to have loaded the driver when code change is underway.
The more useful option is `pending_driver`, which means that reloading will be queued if the driver is *not* loaded by any other users [page ??], but the driver has opened ports, in which case {`ok, pending_driver`} will be returned (a `monitor` option is of course recommended).
If the driver is unloaded (not present in the system), the error code `not_loaded` will be returned. The `reload` option is intended for when the user has already loaded the driver in advance.

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

A number of errors are opaque and can only be interpreted by passing them to the format_error/1 [page ??] function, but some can be interpreted directly:

{***error,linked_in_driver***} The driver with the specified name is an erlang statically linked in driver, which cannot be manipulated with this API.

{***error,inconsistent***} The driver has already been loaded with either other `DriverOptions` or a different *literal* `Path` argument.
This can happen even if a `reload` option is given, if the `DriverOptions` differ from the current.

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

{***error, pending_process***} The driver is loaded by other users [page ??] when the {`reload, pending_driver`} option was given.

{***error, pending_reload***} Driver reload is already requested by another user [page ??] when the {`reload, ReloadOption`} option was given.

{***error, not_loaded_by_this_process***} Appears when the `reload` option is given. The driver `Name` is present in the system, but there is no user [page ??] of it in this process.

{***error, not_loaded***} Appears when the `reload` option is given. The driver `Name` is not in the system. Only drivers loaded by this process can be reloaded.

All other error codes are to be translated by the format_error/1 [page ??] function. Note that calls to `format_error` should be performed from the same running instance of the erlang virtual machine as the error was detected in, due to system dependent behavior concerning error values.

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

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

Types:

- Name = string() | atom()
- OptionList = [Option]
- Option = {monitor, MonitorOption} | kill_ports
- MonitorOption = pending_driver | pending
- Status = unloaded | PendingStatus
- PendingStatus = pending_driver | pending_process
- Ref = ref()
- ErrorAtom = linked_in_driver | not_loaded | not_loaded_by_this_process | permanent

This is the low level function to unload (or decrement reference counts of) a driver. It can be used to force port killing, in much the same way as the driver option `kill_ports` implicitly does, and it can trigger a monitor either due to other users [page ??] still having the driver loaded or that there are open ports using the driver.

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

To allow the user [page ??] that *requests unloading* to wait for *actual unloading* to take place, `monitor` triggers can be specified in much the same way as when loading. As users [page ??] of this function however seldom are interested in more than decrementing the reference counts, monitoring is more seldom needed. If the

kill_ports option is used however, monitor trigging is crucial, as the ports are not guaranteed to have been killed until the driver is unloaded, why a monitor should be triggered for at least the pending_driver case.

The possible monitor messages that can be expected are the same as when using the unloaded option to the monitor/2 [page ??] function.

The function will return one of the following statuses upon success:

{***ok, unloaded***} The driver was immediately unloaded, meaning that the driver name is now free to use by other drivers and, if the underlying OS permits it, the memory occupied by the driver object code is now reclaimed.

The driver can only be unloaded when there are no open ports using it and there are no more users [page ??] requiring it to be loaded.

{***ok, pending_driver***} or {***ok, pending_driver, ref()***} This return value indicates that this call removed the last user [page ??] from the driver, but there are still open ports using it. When all ports are closed and no new users [page ??] have arrived, the driver will actually be reloaded and the name and memory reclaimed.

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

{***ok, pending_process***} or {***ok, pending_process, ref()***} The unload request is registered, but there are still other users [page ??] holding the driver. Note that the term pending_process might refer to the running process, there might be more than one user [page ??] in the same process.

This is a normal, healthy return value if the call was just placed to inform the emulator that you have no further use of the driver. It is actually the most common return value in the most common scenario [page ??] described in the introduction.

The function accepts the following parameters:

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

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

kill_ports Force killing of all ports opened using this driver, with the exit reason driver_unloaded, if you are the *last* user [page ??] of the driver.
If there are other users [page ??] having the driver loaded, this option will have no effect.
If one wants the consistent behavior of killing ports when the last user [page ??] unloads, one should use the driver option kill_ports when loading the driver instead.

{***monitor, MonitorOption***} This option creates a driver monitor if the condition given in MonitorOptions is true. The valid options are:

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

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

The pending_driver MonitorOption is by far the most useful and it has to be used to ensure that the driver has really been unloaded and the ports closed whenever the kill_ports option is used or the driver may have been loaded with the kill_ports driver option.
By using the monitor-triggers in the call to try_unload one can be sure that the monitor is actually added before the unloading is executed, meaning that the monitor will always get properly triggered, which would not be the case if one called erl_ddll:monitor/2 separately.

The function may return several error conditions, of which all are well specified (no opaque values):

{***error, linked_in_driver***} You were trying to unload an erlang statically linked in driver, which cannot be manipulated with this interface (and cannot be unloaded at all).

{***error, not_loaded***} The driver Name is not present in the system.

{***error, not_loaded_by_this_process***} The driver Name is present in the system, but there is no user [page ??] of it in this process.

As a special case, drivers can be unloaded from processes that has done no corresponding call to try_load/3 if, and only if, there are *no users of the driver at all*, which may happen if the process containing the last user dies.

{***error, permanent***} The driver has made itself permanent, in which case it can no longer be manipulated by this interface (much like a statically linked in driver).

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

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

Types:

- Name = string() | atom()
- ErrorDesc = term()

Unloads, or at least dereferences the driver named Name. If the caller is the last user [page ??] of the driver, and there are no more open ports using the driver, the driver will actually get unloaded. In all other cases, actual unloading will be delayed until all ports are closed and there are no remaining users [page ??].

If there are other users [page ??] of the driver, the reference counts of the driver is merely decreased, so that the caller is no longer considered a user of the driver. For usage scenarios, see the description [page ??] in the beginning of this document.

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

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

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

Types:

- Name = string() | atom()
- ErrorDesc = term()

Unloads, or at least dereferences the driver named `Name`. If the caller is the last user [page ??] of the driver, all remaining open ports using the driver will get killed with the reason `driver_unloaded` and the driver will eventually get unloaded.

If there are other users [page ??] of the driver, the reference counts of the driver is merely decreased, so that the caller is no longer considered a user [page ??]. For usage scenarios, see the description [page ??] in the beginning of this document.

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

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

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

Types:

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

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

The driver names are returned as a list of strings rather than a list of atoms for historical reasons.

More information about drivers can be obtained using one of the info [page ??] functions.

`format_error(ErrorDesc) -> string()`

Types:

- ErrorDesc – see below

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

> **Note:**
> Due to peculiarities in the dynamic loading interfaces on different platform, the returned string is only guaranteed to describe the correct error *if format_error/1 is called in the same instance of the erlang virtual machine as the error appeared in* (meaning the same operating system process)!

SEE ALSO

erl_driver(4), driver_entry(4)

erl_prim_loader

Erlang Module

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

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 command line 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 ??], erl_boot_server(3) [page ??]

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".

DATA TYPES

```
ext_binary()
  a binary data object,
  structured according to the Erlang external term format

iodata() = iolist() | binary()

iolist() = [char() | binary() | iolist()]
  a binary is allowed as the tail of the list
```

Exports

`abs(Number) -> int() | float()`

Types:

- Number = number()

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

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

Allowed in guard tests.

`erlang:append_element(Tuple1, Term) -> Tuple2`

Types:

- Tuple1 = Tuple2 = tuple()
- Term = term()

Returns a new tuple which has one element more than `Tuple1`, and contains the elements in `Tuple1` followed by `Term` as the last element. Semantically equivalent to `list_to_tuple(tuple_to_list(Tuple ++ [Term])`, but much faster.

```
> erlang:append_element({one, two}, three).
{one,two,three}
```

`apply(Fun, Args) -> term() | empty()`

Types:

- Fun = fun()
- Args = [term()]

Call a fun, passing the elements in `Args` 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)`.

> **Warning:**
> Earlier, `Fun` could also be given as `{Module, Function}`, equivalent to `apply(Module, Function, Args)`. This usage is deprecated and will stop working in a future release of Erlang/OTP.

`apply(Module, Function, Args) -> term() | empty()`

Types:

- Module = Function = atom()
- Args = [term()]

Returns the result of applying `Function` in `Module` to `Args`. The applied function must be exported from `Module`. The arity of the function is the length of `Args`.

```
> 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 the applied function is not exported. The error handler can be redefined (see process_flag/2 [page ??]). 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` is generated.

`atom_to_list(Atom) -> string()`

Types:

- Atom = atom()

Returns a string which corresponds to the text representation of `Atom`.

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

`binary_to_list(Binary) -> [char()]`

Types:

- Binary = binary()

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

`binary_to_list(Binary, Start, Stop) -> [char()]`

Types:

- Binary = binary()
- Start = Stop = 1..size(Binary)

As `binary_to_list/1`, but returns a list of integers corresponding to the bytes from position `Start` to position `Stop` in `Binary`. Positions in the binary are numbered starting from 1.

`binary_to_term(Binary) -> term()`

Types:

- Binary = ext_binary()

Returns an Erlang term which is the result of decoding the binary object `Binary`, which must be encoded according to the Erlang external term format. See also term_to_binary/1 [page ??].

`erlang:bump_reductions(Reductions) -> void()`

Types:

- Reductions = int()

This implementation-dependent function increments the reduction counter for the calling 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.

`erlang:cancel_timer(TimerRef) -> Time | false`

Types:

- TimerRef = ref()
- Time = int()

Cancels a timer, where `TimerRef` was returned by either erlang:send_after/3 [page ??] or erlang:start_timer/3 [page ??]. If the timer is there to be removed, the function returns the time in milliseconds left until the timer would have expired, otherwise `false` (which means that `TimerRef` was never a timer, that it has already been cancelled, or that it has already delivered its message).

See also erlang:send_after/3 [page ??], erlang:start_timer/3 [page ??], and erlang:read_timer/1 [page ??].

Note: Cancelling a timer does not guarantee that the message has not already been delivered to the message queue.

`check_process_code(Pid, Module) -> bool()`

Types:

- Pid = pid()
- Module = atom()

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

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

See also code(3) [page ??].

`concat_binary(ListOfBinaries)`

Do not use; use list_to_binary/1 [page ??] instead.

`date() -> {Year, Month, Day}`

Types:

- Year = Month = Day = int()

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

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

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

`delete_module(Module) -> true | undefined`

Types:

- Module = atom()

Makes the current code for Module become old code, and deletes all references for this module from the export table. Returns undefined if the module does not exist, otherwise true.

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

Failure: badarg if there is already an old version of Module.

erlang:demonitor(MonitorRef) -> true

Types:

- MonitorRef = ref()

If MonitorRef is a reference which the calling process obtained by calling erlang:monitor/2 [page ??], this monitoring is turned off. If the monitoring is already turned off, nothing happens.

Once erlang:demonitor(MonitorRef) has returned it is guaranteed that no {'DOWN', MonitorRef, _, _, _} message due to the monitor will be placed in the callers message queue in the future. A {'DOWN', MonitorRef, _, _, _} message might have been placed in the callers message queue prior to the call, though. Therefore, in most cases, it is advisable to remove such a 'DOWN' message from the message queue after monitoring has been stopped. erlang:demonitor(MonitorRef, [flush]) [page ??] can be used instead of erlang:demonitor(MonitorRef) if this cleanup is wanted.

> **Note:**
> Prior to OTP release R11B (erts version 5.5) erlang:demonitor/1 behaved completely asynchronous, i.e., the monitor was active until the "demonitor signal" reached the monitored entity. This had one undesirable effect, though. You could never know when you were guaranteed *not* to receive a DOWN message due to the monitor.
>
> Current behavior can be viewed as two combined operations: asynchronously send a "demonitor signal" to the monitored entity and ignore any future results of the monitor.

Failure: It is an error if MonitorRef 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 MonitorRef is a remote reference).

erlang:demonitor(MonitorRef, OptionList) -> true

Types:

- MonitorRef = ref()
- OptionList = [Option]
- Option = flush

`erlang:demonitor(MonitorRef, [])` is equivalent to erlang:demonitor(MonitorRef) [page ??].

Currently the following `Options` are valid:

`flush` Remove (one) {`_`, `MonitorRef`, `_`, `_`, `_`} message, if there is one, from the callers message queue after monitoring has been stopped.

Calling `erlang:demonitor(MonitorRef, [flush])` is equivalent to:

```
erlang:demonitor(MonitorRef),
receive
    {_, MonitorRef, _, _, _} ->
        true
after 0 ->
        true
end
```

> **Note:**
> More options may be added in the future.

Failure: `badarg` if `OptionList` is not a list, or if `Option` is not a valid option, or the same failure as for erlang:demonitor/1 [page ??]

`disconnect_node(Node) -> bool() | ignored`

Types:

- Node = atom()

Forces the disconnection of a node. This will appear to the node `Node` as if the local node has crashed. This BIF is mainly used in the Erlang network authentication protocols. Returns `true` if disconnection succeeds, otherwise `false`. If the local node is not alive, the function returns `ignored`.

`erlang:display(Term) -> true`

Types:

- Term = term()

Prints a text representation of `Term` on the standard output.

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

`element(N, Tuple) -> term()`

Types:

- N = 1..size(Tuple)
- Tuple = tuple()

Returns the `N`th element (numbering from 1) of `Tuple`.

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

Allowed in guard tests.

`erase() -> [{Key, Val}]`

Types:

- Key = Val = term()

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) -> Val | undefined`

Types:

- Key = Val = term()

Returns the value `Val` associated with `Key` and deletes it from the process dictionary. Returns `undefined` if no value is associated with `Key`.

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

`erlang:error(Reason)`

Types:

- Reason = term()

Stops the execution of the calling 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.

```
> catch erlang:error(foobar).
{'EXIT',{foobar,[{erl_eval,do_apply,5},
                 {erl_eval,expr,5},
                 {shell,exprs,6},
                 {shell,eval_loop,3}]}}
```

`erlang:error(Reason, Args)`

Types:

- Reason = term()
- Args = [term()]

Stops the execution of the calling 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)`

Types:

- Reason = term()

Stops the execution of the calling process with the exit reason Reason, where Reason is any 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) -> true`

Types:

- Pid = pid()
- Reason = term()

Sends an exit signal with exit reason Reason to the process Pid.

The following behavior apply if Reason is any term except `normal` or `kill`:

If Pid is not trapping exits, Pid itself will exit with exit reason Reason. If Pid is trapping exits, the exit signal is transformed into a message {'EXIT', From, Reason} and delivered to the message queue of Pid. From is the pid of the process which sent the exit signal. See also process_flag/2 [page ??].

If Reason is the atom `normal`, Pid will not exit. If it is trapping exits, the exit signal is transformed into a message {'EXIT', From, normal} and delivered to its message queue.

If 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 exit reason `killed`.

`erlang:fault(Reason)`

Types:

- Reason = term()

Stops the execution of the calling process with the reason Reason. This is an old equivalent to erlang:error(Reason) [page ??].

`erlang:fault(Reason, Args)`

Types:

- Reason = term()
- Args = [term()]

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

`float(Number) -> float()`

Types:

- Number = number()

Returns a float by converting Number to a float.

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

Allowed in guard tests.

> **Note:**
> Note that if 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 ??] instead.
>
> When float/1 is used in an expression in a guard, such as 'float(A) == 4.0', it converts a number as described above.

`float_to_list(Float) -> string()`

Types:

- Float = float()

Returns a string which corresponds to the text representation of Float.

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

`erlang:fun_info(Fun) -> [{Item, Info}]`

Types:

- Fun = fun()
- Item, Info – see below

Returns a list containing information about the fun Fun. Each element of the list is a tuple. The order of the tuples is not defined, and more tuples may be added in a future release.

> **Warning:**
> This BIF is mainly intended for debugging, but it can occasionally be useful in library functions that might need to verify, for instance, the arity of a fun.

There are two types of funs with slightly different semantics:

A fun created by `fun M:F/A` is called an *external* fun. Calling it will always call the function `F` with arity `A` in the latest code for module `M`. Note that module `M` does not even need to be loaded when the fun `fun M:F/A` is created.

All other funs are called *local*. When a local fun is called, the same version of the code that created the fun will be called (even if newer version of the module has been loaded).

The following elements will always be present in the list for both local and external funs:

`{type, Type}` `Type` is either `local` or `external`.

`{module, Module}` `Module` (an atom) is the module name.

If `Fun` is a local fun, `Module` is the module in which the fun is defined.

If `Fun` is an external fun, `Module` is the module that the fun refers to.

`{name, Name}` `Name` (an atom) is a function name.

If `Fun` is a local fun, `Name` is the name of the local function that implements the fun. (This name was generated by the compiler, and is generally only of informational use. As it is a local function, it is not possible to call it directly.) If no code is currently loaded for the fun, `[]` will be returned instead of an atom.

If `Fun` is an external fun, `Name` is the name of the exported function that the fun refers to.

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

`{env, Env}` `Env` (a list) is the environment or free variables for the fun. (For external funs, the returned list is always empty.)

The following elements will only be present in the list if `Fun` is local:

`{pid, Pid}` `Pid` is the pid of the process that originally created the fun.

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

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

`{new_uniq, Uniq}` `Uniq` (a binary) is a unique value for this fun.

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

`erlang:fun_info(Fun, Item) -> {Item, Info}`

Types:

- Fun = fun()
- Item, Info – see below

Returns information about `Fun` as specified by `Item`, in the form `{Item,Info}`.

For any fun, `Item` can be any of the atoms `module`, `name`, `arity`, or `env`.

For a local fun, `Item` can also be any of the atoms `index`, `new_index`, `new_uniq`, `uniq`, and `pid`. For an external fun, the value of any of these items is always the atom `undefined`.

See erlang:fun_info/1 [page ??].

`erlang:fun_to_list(Fun) -> string()`

Types:

- Fun = fun()

Returns a string which corresponds to the text representation of Fun.

`erlang:function_exported(Module, Function, Arity) -> bool()`

Types:

- Module = Function = atom()
- Arity = int()

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.

`garbage_collect() -> true`

Forces an immediate garbage collection of the currently executing process. The function should not be used, unless it has been noticed – or there are 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) -> bool()`

Types:

- Pid = 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() -> [{Key, Val}]`

Types:

- Key = Val = term()

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

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

`get(Key) -> Val | undefined`

Types:

- Key = Val = term()

Returns the value `Val` associated with `Key` in the process dictionary, or `undefined` if `Key` does not exist.

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

`erlang:get_cookie() -> Cookie | nocookie`

Types:

- Cookie = atom()

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

`get_keys(Val) -> [Key]`

Types:

- Val = Key = term()

Returns a list of keys which are associated with the value `Val` 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() -> [{Module, Function, Arity | Args}]`

Types:

- Module = Function = atom()
- Arity = int()
- Args = [term()]

Get the call stack backtrace (*stacktrace*) of the last exception in the calling process as a list of `{Module,Function,Arity}` tuples. The `Arity` field in the first tuple may be the argument list of that function call 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 as the `catch` operator returns, for example:

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

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

`group_leader() -> GroupLeader`

Types:

- GroupLeader = pid()

Returns the 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*. All IO from the group is channeled to the group leader. When a new process is spawned, it gets the same group leader as the spawning process. Initially, at system start-up, `init` is both its own group leader and the group leader of all processes.

`group_leader(GroupLeader, Pid) -> true`

Types:

- GroupLeader = Pid = pid()

Sets the group leader of `Pid` to `GroupLeader`. Typically, this is used when a processes started from a certain shell should have another group leader than `init`.

See also group_leader/0 [page ??].

`halt()`

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

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

`halt(Status)`

Types:

- Status = int()>=0 | string()

`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) -> Hash`

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) -> term()`

Types:

- List = [term()]

Returns the head of `List`, that is, the first element.

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

Allowed in guard tests.

Failure: `badarg` if `List` is the empty list [].

`erlang:hibernate(Module, Function, Args)`

Types:

- Module = Function = atom()
- Args = [term()]

Puts the calling 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 `Args` 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` does is the following. It discards the call stack for the process. Then it garbage collects the process. After the garbage collection, all live data is in one continuous heap. The heap is then shrunken to the exact same size as the live data which it holds (even if that size is 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.

Note that emptying the call stack means that any surrounding `catch` is removed and has to be re-inserted after hibernation. One effect of this is that processes started using `proc_lib` (also indirectly, such as `gen_server` processes), should use [proc_lib:hibernate/3] instead to ensure that the exception handler continues to work when the process wakes up.

`erlang:info(Type) -> Res`

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

`integer_to_list(Integer) -> string()`

Types:

- Integer = int()

Returns a string which corresponds to the text representation of `Integer`.

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

`erlang:integer_to_list(Integer, Base) -> string()`

Types:

- Integer = int()
- Base = 2..36

Returns a string which corresponds to the text representation of `Integer` in base `Base`.

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

`iolist_to_binary(IoListOrBinary) -> binary()`

Types:

- IoListOrBinary = iolist() | binary()

Returns a binary which is made from the integers and binaries in `IoListOrBinary`.

```
> Bin1 = <<1,2,3>>.
<<1,2,3>>
> Bin2 = <<4,5>>.
<<4,5>>
> Bin3 = <<6>>.
<<6>>
> iolist_to_binary([Bin1,1,[2,3,Bin2],4|Bin3]).
<<1,2,3,1,2,3,4,5,4,6>>
```

`iolist_size(Item) -> int()`

Types:

- Item = iolist() | binary()

Returns an integer which is the size in bytes of the binary that would be the result of `iolist_to_binary(Item)`.

```
> iolist_size([1,2|<<3,4>>]).
4
```

`is_alive() -> bool()`

Returns `true` if the local node is alive; that is, if the node can be part of a distributed system. Otherwise, it returns `false`.

`is_atom(Term) -> bool()`

Types:

- Term = term()

Returns `true` if `Term` is an atom; otherwise returns `false`.

Allowed in guard tests.

`is_binary(Term) -> bool()`

Types:

- Term = term()

Returns `true` if `Term` is a binary; otherwise returns `false`.

Allowed in guard tests.

`is_boolean(Term) -> bool()`

Types:

- Term = term()

Returns `true` if `Term` is either the atom `true` or the atom `false` (i.e. a boolean); otherwise returns `false`.

Allowed in guard tests.

`erlang:is_builtin(Module, Function, Arity) -> bool()`

Types:

- Module = Function = atom()
- Arity = int()

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()

Returns `true` if `Term` is a floating point number; otherwise returns `false`.

Allowed in guard tests.

`is_function(Term) -> bool()`

Types:

- Term = term()

Returns `true` if `Term` is a fun; otherwise returns `false`.

Allowed in guard tests.

`is_function(Term, Arity) -> bool()`

Types:

- Term = term()
- Arity = int()

Returns `true` if `Term` is a fun that can be applied with `Arity` number of arguments; otherwise returns `false`.

Allowed in guard tests.

> **Warning:**
> Currently, `is_function/2` will also return `true` if the first argument is a tuple fun (a tuple containing two atoms). In a future release, tuple funs will no longer be supported and `is_function/2` will return `false` if given a tuple fun.

`is_integer(Term) -> bool()`

Types:

- Term = term()

Returns `true` if `Term` is an integer; otherwise returns `false`.

Allowed in guard tests.

`is_list(Term) -> bool()`

Types:

- Term = term()

Returns `true` if `Term` is a list with zero or more elements; otherwise returns `false`.

Allowed in guard tests.

`is_number(Term) -> bool()`

Types:

- Term = term()

Returns `true` if `Term` is either an integer or a floating point number; otherwise returns `false`.

Allowed in guard tests.

`is_pid(Term) -> bool()`

Types:

- Term = term()

Returns `true` if `Term` is a pid (process identifier); otherwise returns `false`.

Allowed in guard tests.

`is_port(Term) -> bool()`

Types:

- Term = term()

Returns `true` if `Term` is a port identifier; otherwise returns `false`.

Allowed in guard tests.

`is_process_alive(Pid) -> bool()`

Types:

- Pid = pid()

`Pid` must refer to a process at the local node. Returns `true` if the process exists and is alive, that is, has not exited. Otherwise, returns `false`.

`is_record(Term, RecordTag) -> bool()`

Types:

- Term = term()
- RecordTag = 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 and the size of the tuple will not be verified.

Allowed in guard tests, if `RecordTag` is a literal atom.

`is_record(Term, RecordTag, Size) -> bool()`

Types:

- Term = term()
- RecordTag = atom()
- Size = int()

`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`.

Allowed in guard tests, provided that `RecordTag` is a literal atom and `Size` is a literal integer.

> **Note:**
> This BIF is documented for completeness. In most cases `is_record/2` should be used.

`is_reference(Term) -> bool()`

Types:

- Term = term()

Returns `true` if `Term` is a reference; otherwise returns `false`.

Allowed in guard tests.

`is_tuple(Term) -> bool()`

Types:

- Term = term()

Returns `true` if `Term` is a tuple; otherwise returns `false`.

Allowed in guard tests.

`length(List) -> int()`

Types:

- List = [term()]

Returns the length of `List`.

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

Allowed in guard tests.

`link(Pid) -> true`

Types:

- Pid = pid() | port()

Creates a link between the calling process and another 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`.

If `Pid` does not exist, the behavior of the BIF depends on if the calling process is trapping exits or not (see process_flag/2 [page ??]):

- If the calling process is not trapping exits, and checking `Pid` is cheap – that is, if `Pid` is local – `link/1` fails with reason `noproc`.
- Otherwise, if the calling process is trapping exits, and/or `Pid` is remote, `link/1` returns `true`, but an exit signal with reason `noproc` is sent to the calling process.

`list_to_atom(String) -> atom()`

Types:

- String = string()

Returns the atom whose text representation is `String`.

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

`list_to_binary(IoList) -> binary()`

Types:

- IoList = iolist()

Returns a binary which is made from the integers and binaries in `IoList`.

```
> Bin1 = <<1,2,3>>.
<<1,2,3>>
> Bin2 = <<4,5>>.
<<4,5>>
> Bin3 = <<6>>.
<<6>>
> list_to_binary([Bin1,1,[2,3,Bin2],4|Bin3]).
<<1,2,3,1,2,3,4,5,4,6>>
```

`list_to_existing_atom(String) -> atom()`

Types:

- String = string()

Returns the atom whose text representation is `String`, but only if there already exists such atom.

Failure: `badarg` if there does not already exist an atom whose text representation is `String`.

`list_to_float(String) -> float()`

Types:

- String = string()

Returns the float whose text representation is `String`.

```
> list_to_float("2.2017764e+0").
2.20178
```

Failure: `badarg` if `String` contains a bad representation of a float.

`list_to_integer(String) -> int()`

Types:

- String = string()

Returns an integer whose text representation is `String`.

```
> list_to_integer("123").
123
```

Failure: `badarg` if `String` contains a bad representation of an integer.

`erlang:list_to_integer(String, Base) -> int()`

Types:

- String = string()
- Base = 2..36

Returns an integer whose text representation in base `Base` is `String`.

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

Failure: badarg if String contains a bad representation of an integer.

`list_to_pid(String) -> pid()`

Types:

- String = string()

Returns a pid whose text representation is String.

> **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 String contains a bad representation of a pid.

`list_to_tuple(List) -> tuple()`

Types:

- List = [term()]

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

```
> list_to_tuple([share, ['Ericsson_B', 163]]).
{share, ['Ericsson_B', 163]}
```

`load_module(Module, Binary) -> {module, Module} | {error, Reason}`

Types:

- Module = atom()
- Binary = binary()
- Reason = badfile | not_purged | badfile

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}, 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.

badfile The object code contains code for another module than Module

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

`erlang:loaded() -> [Module]`

Types:

- Module = atom()

Returns a list of all loaded Erlang modules (current and/or old code), including preloaded modules.

See also code(3) [page ??].

`erlang:localtime() -> {Date, Time}`

Types:

- Date = {Year, Month, Day}
- Time = {Hour, Minute, Second}
- Year = Month = Day = Hour = Minute = Second = int()

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({Date1, Time1}) -> {Date2, Time2}`

Types:

- Date1 = Date2 = {Year, Month, Day}
- Time1 = Time2 = {Hour, Minute, Second}
- Year = Month = Day = Hour = Minute = Second = int()

Converts local date and time to Universal Time Coordinated (UTC), if this is supported by the underlying OS. Otherwise, no conversion is done and {`Date1, Time1`} is returned.

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

Failure: `badarg` if `Date1` or `Time1` do not denote a valid date or time.

`erlang:localtime_to_universaltime({Date1, Time1}, IsDst) -> {Date2, Time2}`

Types:

- Date1 = Date2 = {Year, Month, Day}
- Time1 = Time2 = {Hour, Minute, Second}
- Year = Month = Day = Hour = Minute = Second = int()
- IsDst = true | false | undefined

Converts local date and time 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 `{Date1, Time1}` 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({Date1, Time1})`.

```
> 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}}
```

Failure: `badarg` if `Date1` or `Time1` do not denote a valid date or time.

`make_ref() -> 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) -> tuple()`

Types:

- Arity = int()
- InitialValue = term()

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

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

`erlang:md5(Data) -> Digest`

Types:

- Data = iodata()
- 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.

`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 = iodata()
- Context = NewContext = binary()

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

`erlang:memory() -> [{Type, Size}]`

Types:

- Type, Size – see below

Returns a list containing information about memory dynamically allocated by the Erlang emulator. Each element of the list is a tuple `{Type, Size}`. The first element `Type`is an atom describing memory type. The second element `Size`is memory size in bytes. A description of each memory type follows:

`total` The total amount of memory currently allocated, which is the same as the sum of memory size for `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 [instrument(3)] and/or [erl(1)].

> **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(Type | [Type]) -> Size | [{Type, Size}]`

Types:

- Type, Size – see below

Returns the memory size in bytes allocated for memory of type `Type`. The argument can also be given as a list of `Type` atoms, in which case a corresponding list of `{Type, Size}` tuples is returned.

See erlang:memory/0 [page ??].

Failure: `badarg` if the emulator is not run with instrumentation when `Type == maximum`.

`module_loaded(Module) -> bool()`

Types:

- Module = atom()

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 the code server (see code(3) [page ??]) and should not be used elsewhere.

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

Types:

- Type = process
- Item = pid() | {RegName, Node} | RegName
- RegName = atom()
- Node = node()
- MonitorRef = reference()

The calling 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.

`Item` can be:

`pid()` The pid of the process to monitor.

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

`RegName` The process locally registered as `RegName` will be monitored.

> **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', MonitorRef, Type, Object, Info}`

where `MonitorRef` and `Type` are the same as described above, and:

`Object` A reference to the monitored object:

- the pid of the monitored process, if `Item` was specified as a pid.
- `{RegName, Node}`, if `Item` was specifed as `{RegName, Node}`.
- `{RegName, Node}`, if `Item` was specified as `RegName`. `Node` will in this case be the name of the local node (`node()`).

`Info` Either the exit reason of the process, `noproc` (non-existing process), or `noconnection` (no connection to `Node`).

> **Note:**
> If/when `erlang:monitor/2` is extended (e.g. to handle other item types than `process`), other possible values for `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/1 [page ??] is called.

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 as many, 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) -> true`

Types:

- Node = node()
- Flag = bool()

Monitors the status of the node `Node`. If `Flag` is `true`, monitoring is turned on; if `Flag` is `false`, monitoring is turned off.

Making several calls to `monitor_node(Node, true)` for the same `Node` is not an error; it results in as many, completely independent, monitorings.

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.

Failure: `badarg`if the local node is not alive.

`erlang:monitor_node(Node, Flag, Options) -> true`

Types:

- Node = node()
- Flag = bool()
- Options = [Option]
- Option = allow_passive_connect

Behaves as `monitor_node/2` except that it allows an extra option to be given, namely `allow_passive_connect`. The option allows the bif to wait the normal net connection timeout for the *monitored node* to connect itself, even if it cannot be actively connected from this node (i.e. it is blocked). The state where this might be useful can only be achieved by using the kernel option `dist_auto_connect once`. If that kernel option is not used, the `allow_passive_connect` option has no effect.

> **Note:**
> The `allow_passive_connect` option is used internally and is seldom needed in applications where the network topology and the kernel options in effect is known in advance.

Failure: `badarg` if the local node is not alive or the option list is malformed.

`node() -> Node`

Types:

- Node = node()

Returns the name of the local node. If the node is not alive, `nonode@nohost` is returned instead.

Allowed in guard tests.

`node(Arg) -> Node`

Types:

- Arg = pid() | port() | ref()
- Node = node()

Returns the node where `Arg` is located. `Arg` can be a pid, a reference, or a port. If the local node is not alive, `nonode@nohost` is returned.

Allowed in guard tests.

`nodes() -> Nodes`

Types:

- Nodes = [node()]

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

`nodes(Arg | [Arg]) -> Nodes`

Types:

- Arg = visible | hidden | connected | this | known

- Nodes = [node()]

Returns a list of nodes according to argument given. The result returned when the argument is a list, is the list of nodes satisfying the disjunction(s) of the list elements.

`Arg` can be any of the following:

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

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

`connected` All nodes connected to this node.

`this` This node.

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

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

If the local node is not alive, `nodes(this) == nodes(known) == [nonode@nohost]`, for any other `Arg` the empty list [] is returned.

`now() -> {MegaSecs, Secs, MicroSecs}`

Types:

- MegaSecs = Secs = MicroSecs = int()

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) -> port()`

Types:

- PortName = {spawn, Command} | {fd, In, Out}
- Command = string()
- In = Out = int()
- PortSettings = [Opt]
- Opt = {packet, N} | stream | {line, L} | {cd, Dir} | {env, Env} | exit_status | use_stdio | nouse_stdio | stderr_to_stdout | in | out | binary | eof
- N = 1 | 2 | 4
- L = int()
- Dir = string()
- Env = [{Name, Val}]
- Name = string()
- Val = string() | false

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.

{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 settings 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, L} 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).

L 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, L} 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, Env} This is only valid for {spawn, Command}. The environment of the started process is extended using the environment specifications in Env.

Env should be a list of tuples {Name, Val}, where Name is the name of an environment variable, and Val is the value it is to have in the spawned port process. Both Name and Val must be strings. The one exception is Val 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 `use_stdio`. 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 IO 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: 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`}. See `file(3)` for possible values of `PosixCode`.

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) -> Hash`

Types:

- Term = term()
- Range = 1..2^32
- Hash = 1..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]) -> Hash`

Types:

- Term = term()

- Range = 1..2^32
- Hash = 0..Range-1

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) -> string()`

Types:

- Pid = pid()

Returns a string which corresponds to the text representation of `Pid`.

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

`port_close(Port) -> true`

Types:

- Port = port() | atom()

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).

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 calling 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 calling 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}`.

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

`port_command(Port, Data) -> true`

Types:

- Port = port() | atom()
- Data = iodata()

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).

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 calling process is not the port owner, the *port owner* fails with `badsig`. The port owner fails with `badsig` also if `Data` is not a valid IO 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}}`.

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

`port_connect(Port, Pid) -> true`

Types:

- Port = port() | atom()
- Pid = 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`.

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 calling 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 calling process is not the port owner, the *port owner* fails with `badsig`. The port owner fails with `badsig` also if `Pid` is not an existing 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}}`.

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

`port_control(Port, Operation, Data) -> Res`

Types:

- Port = port() | atom()
- Operation = int()
- Data = Res = iodata()

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 an open 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_call(Port, Operation, Data) -> term()`

Types:

- Port = port() | atom()
- Operation = int()
- Data = term()

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 a port identifier, 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 an open 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, Info}] | undefined`

Types:

- Port = port() | atom()
- Item, Info – see below

Returns a list containing tuples with information about the `Port`, or `undefined` if the port is not open. The order of the tuples is not defined, nor are all the tuples mandatory.

`{registered_name, RegName}` `RegName` (an 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` (an integer) 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, Pids}` `Pids` is a list of pids to which processes the port is linked.

`{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.

Failure: badarg if Port is not a local port.

erlang:port_info(Port, Item) -> {Item, Info} | undefined | []

Types:

- Port = port() | atom()
- Item, Info – see below

Returns information about Port as specified by Item, or undefined if the port is not open. Also, if Item == registered_name and the port has no registered name, [] is returned.

For valid values of Item, and corresponding values of Info, see erlang:port_info/1 [page ??].

Failure: badarg if Port is not a local port.

erlang:port_to_list(Port) -> string()

Types:

- Port = port()

Returns a string which corresponds to the text representation of 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:ports() -> [port()]

Returns a list of all ports on the local node.

pre_loaded() -> [Module]

Types:

- Module = atom()

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) -> void()

Types:

- Pid = pid()
- Type = backtrace

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 call 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, Value) -> OldValue

Types:

- Flag, Value, OldValue – see below

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.

See also exit/2 [page ??].

`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 calling 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.

`process_flag(Pid, Flag, Value) -> OldValue`

Types:

- Pid = pid()
- Flag, Value, OldValue – see below

Sets certain flags for the process `Pid`, in the same manner as process_flag/2 [page ??]. 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 local process.

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

Types:

- Pid = pid()
- Item, Info – see below

Returns a list containing tuples with information about the process `Pid`, or `undefined` if the process is not alive. The order of the tuples is 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 IO of the process.

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

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

`{links, Pids}` `Pids` 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 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 local process.

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

Types:

- Pid = pid()
- Item, Info – see below

Returns information about the process Pid as specified by Item, or undefined if the process is not alive. Also, if Item == registered_name and the process has no registered name, [] is returned.

The value of Item, and corresponding value of Info, can be any of the values specified for process_info/1 [page ??].

In addition to the above, also the following items – with corresponding values – are allowed:

{backtrace, Bin} The binary Bin 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.

{last_calls, false|Calls} The value is false if call saving is not active for the process (see process_flag/3 [page ??]). If call saving is active, a list is returned, in which the last element is the most recent called.

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

{monitored_by, Pids} A list of pids that are monitoring the process (with erlang:monitor/2).

{monitors, Monitors} 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, {RegName, Node}}.

Note however, that not all implementations support every one of the above Items.

Failure: badarg if Pid is not a local process.

processes() -> [pid()]

Returns a list of all processes on the local node.

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

purge_module(Module) -> void()

Types:

- Module = atom()

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 the module.

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

Failure: `badarg` if there is no old code for `Module`.

`put(Key, Val) -> OldVal | undefined`

Types:

- Key = Val = OldVal = term()

Adds a new `Key` to the process dictionary, associated with the value `Val`, and returns `undefined`. If `Key` already exists, the old value is deleted and replaced by `Val` and the function returns the old value.

> **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)`

Types:

- Class = error | exit | throw
- Reason = term()
- Stacktrace = [{Module, Function, Arity | Args} | {Fun, Args}]
- Module = Function = atom()
- Arity = int()
- Args = [term()]
- Fun = [fun()]

Stops the execution of the calling process with an exception of given class, reason and call stack backtrace (*stacktrace*).

> **Warning:**
> This BIF is intended for debugging and for use in the Erlang operating system. In general, it should be avoided in applications, unless you know very well what you are doing.

`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, Arity | Args`} where `Module` and `Function` are atoms and the third element is an integer arity or an argument list. The stacktrace may also contain {`Fun, Args`} tuples where `Fun` is a local fun and `Args` is an argument list.

The stacktrace is used as the exception stacktrace for the calling process; it will be truncated to the current maximum stacktrace depth.

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(TimerRef) -> int() | false`

Types:

- TimerRef = ref()

`TimerRef` is a timer reference returned by erlang:send_after/3 [page ??] or erlang:start_timer/3 [page ??]. If the timer is active, the function returns the time in milliseconds left until the timer will expire, otherwise `false` (which means that `TimerRef` was never a timer, that it has been cancelled, or that it has already delivered its message).

See also erlang:send_after/3 [page ??], erlang:start_timer/3 [page ??], and erlang:cancel_timer/1 [page ??].

`erlang:ref_to_list(Ref) -> string()`

Types:

- Ref = ref()

Returns a string which corresponds to the text representation of `Ref`.

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

`register(RegName, Pid | Port) -> true`

Types:

- RegName = atom()
- Pid = pid()
- Port = port()

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

```
> register(db, Pid).
true
```

Failure: `badarg` if `Pid` is not an existing, local process or port, if `RegName` is already in use, if the process or port is already registered (already has a name), or if `RegName` is the atom `undefined`.

`registered() -> [RegName]`

Types:

- RegName = atom()

Returns a list of names which have been registered using register/2 [page ??].

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

`erlang:resume_process(Pid) -> true`

Types:

- Pid = pid()

Resume a suspended process `Pid`.

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

Failure: `badarg` if `Pid` does not exist.

`round(Number) -> int()`

Types:

- Number = number()

Returns an integer by rounding `Number`.

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

Allowed in guard tests.

`self() -> pid()`

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

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

Allowed in guard tests.

`erlang:send(Dest, Msg) -> Msg`

Types:

- Dest = pid() | port() | RegName | {RegName, Node}
- Msg = term()
- RegName = atom()
- Node = node()

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 {RegName, Node} for a registered name at another node.

erlang:send(Dest, Msg, [Option]) -> Res

Types:

- Dest = pid() | port() | RegName | {RegName, Node}
- RegName = atom()
- Node = node()
- Msg = term()
- Option = nosuspend | noconnect
- Res = ok | nosuspend | noconnect

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

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.

> **Warning:**
> As with erlang:send_nosuspend/2,3: Use with extreme care!

erlang:send_after(Time, Dest, Msg) -> TimerRef

Types:

- Time = int()
- 0 <= Time <= 4294967295
- Dest = pid() | RegName
- LocalPid = pid() (of a process, alive or dead, on the local node)
- Msg = term()
- TimerRef = ref()

Starts a timer which will send the message Msg to Dest after Time milliseconds.

If Dest 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.

If Dest is a pid, the timer will be automatically canceled if the process referred to by the pid is not alive, or when the process exits. This feature was introduced in erts version 5.4.11. Note that timers will not be automatically canceled when Dest is an atom.

See also erlang:start_timer/3 [page ??], erlang:cancel_timer/1 [page ??], and erlang:read_timer/1 [page ??].

Failure: badarg if the arguments does not satisfy the requirements specified above.

`erlang:send_nosuspend(Dest, Msg) -> bool()`

Types:

- Dest = pid() | port() | RegName | {RegName, Node}
- RegName = atom()
- Node = node()
- Msg = term()

The same as erlang:send(Dest, Msg, [nosuspend]) [page ??], but returns `true` if the message was sent and `false` if the message was not sent because the sender would have had to be 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.

> **Warning:**
> Use with extreme care!

`erlang:send_nosuspend(Dest, Msg, Options) -> bool()`

Types:

- Dest = pid() | port() | RegName | {RegName, Node}
- RegName = atom()
- Node = node()
- Msg = term()
- Option = noconnect

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

This function behaves like erlang:send_nosuspend/2) [page ??], 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 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.

Warning:
Use with extreme care!

`erlang:set_cookie(Node, Cookie) -> true`

Types:

- Node = node()
- Cookie = atom()

Sets the magic cookie of `Node` to the atom `Cookie`. If `Node` is the local node, the function also sets the cookie of all other unknown nodes to `Cookie` (see [Distributed Erlang] in the Erlang Reference Manual).

Failure: `function_clause` if the local node is not alive.

`setelement(Index, Tuple1, Value) -> Tuple2`

Types:

- Index = 1..size(Tuple1)
- Tuple1 = Tuple2 = tuple()
- Value = term()

Returns a tuple which is a copy of the argument `Tuple1` 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}
```

`size(Item) -> int()`

Types:

- Item = tuple() | binary()

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.

`spawn(Fun) -> pid()`

Types:

- Fun = fun()

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

`spawn(Node, Fun) -> pid()`

Types:

- Node = node()
- Fun = fun()

Returns the pid of a new process started by the application of `Fun` to the empty list `[]` on `Node`. If `Node` does not exist, a useless pid is returned. Otherwise works like spawn/3 [page ??].

`spawn(Module, Function, Args) -> pid()`

Types:

- Module = Function = atom()
- Args = [term()]

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

`error_handler:undefined_function(Module, Function, Args)` is evaluated by the new process if `Module:Function/Arity` does not exist (where `Arity` is the length of `Args`). The error handler can be redefined (see process_flag/2 [page ??]). 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>
```

`spawn(Node, Module, Function, ArgumentList) -> pid()`

Types:

- Node = node()
- Module = Function = atom()
- Args = [term()]

Returns the pid of a new process started by the application of `Module:Function` to `Args` on `Node`. If `Node` does not exists, a useless pid is returned. Otherwise works like spawn/3 [page ??].

`spawn_link(Fun) -> pid()`

Types:

- Fun = fun()

Returns the pid of a new process started by the application of Fun to the empty list []. A link is created between the calling process and and the new process, atomically. Otherwise works like spawn/3 [page ??].

`spawn_link(Node, Fun) ->`

Types:

- Node = node()
- Fun = fun()

Returns the pid of a new process started by the application of Fun to the empty list [] on Node. A link is created between the calling process and and the new process, atomically. If Node does not exist, a useless pid is returned (and due to the link, an exit signal with exit reason noconnection will be received). Otherwise works like spawn/3 [page ??].

`spawn_link(Module, Function, Args) -> pid()`

Types:

- Module = Function = atom()
- Args = [term()]

Returns the pid of a new process started by the application of Module:Function to Args. A link is created betwen the calling process and the new process, atomically. Otherwise works like spawn/3 [page ??].

`spawn_link(Node, Module, Function, Args) -> pid()`

Types:

- Node = node()
- Module = Function = atom()
- Args = [term()]

Returns the pid of a new process started by the application of Module:Function to Args on Node. A link is created betwen the calling process and the new process, atomically. If Node does not exist, a useless pid is returned (and due to the link, an exit signal with exit reason noconnection will be received). Otherwise works like spawn/3 [page ??].

`erlang:spawn_monitor(Fun) -> {pid(),reference()}`

Types:

- Fun = fun()

Returns the pid of a new process started by the application of Fun to the empty list [] and reference for a monitor created to the new process. Otherwise works like spawn/3 [page ??].

`erlang:spawn_monitor(Module, Function, Args) -> {pid(),reference()}`

Types:

- Module = Function = atom()
- Args = [term()]

A new process is started by the application of `Module:Function` to `Args`, and the process is monitored at the same time. Returns the pid and a reference for the monitor. Otherwise works like spawn/3 [page ??].

`spawn_opt(Fun, [Option]) -> pid() | {pid(),reference()}`

Types:

- Fun = fun()
- Option = link | monitor | {priority, Level} | {fullsweep_after, Number} | {min_heap_size, Size}
- Level = low | normal | high
- Number = int()
- Size = int()

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

If the option `monitor` is given, the newly created process will be monitored and both the pid and reference for the monitor will be returned.

`spawn_opt(Node, Fun, [Option]) -> pid()`

Types:

- Node = node()
- Fun = fun()
- Option = link | {priority, Level} | {fullsweep_after, Number} | {min_heap_size, Size}
- Level = low | normal | high
- Number = int()
- Size = int()

Returns the pid of a new process started by the application of `Fun` to the empty list `[]` on `Node`. If `Node` does not exist, a useless pid is returned. Otherwise works like spawn_opt/4 [page ??].

`spawn_opt(Module, Function, Args, [Option]) -> pid() | {pid(),reference()}`

Types:

- Module = Function = atom()
- Args = [term()]
- Option = link | monitor | {priority, Level} | {fullsweep_after, Number} | {min_heap_size, Size}
- Level = low | normal | high
- Number = int()
- Size = int()

Works exactly like spawn/3 [page ??], except that an extra option list is given when creating the process.

If the option `monitor` is given, the newly created process will be monitored and both the pid and reference for the monitor will be returned.

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

`monitor` Monitor the new process (just like erlang:monitor/2 [page ??] 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} This option is only useful for performance tuning. In general, you should not use this option unless you know that there is problem with execution times and/or memory consumption, and you should measure to make sure that the option improved matters.

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.

The fullsweep_after option makes it possible to 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 binaries that are no longer used should 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, one might want to preserve memory by setting Number to zero. (The value may be set globally, see erlang:system_flag/2 [page ??].)

{min_heap_size, Size} This option is only useful for performance tuning. In general, you should not use this option unless you know that there is problem with execution times and/or memory consumption, and you should measure to make sure that the option improved matters.

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, Args, [Option]) -> pid()
```

Types:

- Node = node()
- Module = Function = atom()
- Args = [term()]
- Option = link | {priority, Level} | {fullsweep_after, Number} | {min_heap_size, Size}
- Level = low | normal | high
- Number = int()
- Size = int()

Returns the pid of a new process started by the application of Module:Function to Args on Node. If Node does not exist, a useless pid is returned. Otherwise works like spawn_opt/4 [page ??].

```
split_binary(Bin, Pos) -> {Bin1, Bin2}
```

Types:

- Bin = Bin1 = Bin2 = binary()
- Pos = 1..size(Bin)

Returns a tuple containing the binaries which are the result of splitting `Bin` into two parts at position `Pos`. This is not a destructive operation. After the operation, there will be three binaries altogether.

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

`erlang:start_timer(Time, Dest, Msg) -> TimerRef`

Types:

- Time = int()
- 0 <= Time <= 4294967295
- Dest = LocalPid | RegName
- LocalPid = pid() (of a process, alive or dead, on the local node)
- RegName = atom()
- Msg = term()
- TimerRef = ref()

Starts a timer which will send the message `{timeout, TimerRef, Msg}` to `Dest` after `Time` milliseconds.

If `Dest` 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.

If `Dest` is a pid, the timer will be automatically canceled if the process referred to by the pid is not alive, or when the process exits. This feature was introduced in erts version 5.4.11. Note that timers will not be automatically canceled when `Dest` is an atom.

See also erlang:send_after/3 [page ??], erlang:cancel_timer/1 [page ??], and erlang:read_timer/1 [page ??].

Failure: `badarg` if the arguments does not satisfy the requirements specified above.

`statistics(Type) -> Res`

Types:

- Type, Res – see below

Returns information about the system as specified by `Type`:

`context_switches` Returns `{ContextSwitches, 0}`, where `ContextSwitches` is the total number of context switches since the system started.

`exact_reductions` Returns {`Total_Exact_Reductions`, `Exact_Reductions_Since_Last_Call`}.

NOTE: `statistics(exact_reductions)` is a more expensive operation than statistics(reductions) [page ??] especially on an Erlang machine with smp support.

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

`io` Returns {{`input`, `Input`}, {`output`, `Output`}}, where `Input` is the total number of bytes received through ports, and `Output` is the total number of bytes output to ports.

`reductions` Returns {`Total_Reductions`, `Reductions_Since_Last_Call`}.

NOTE: From erts version 5.5 (OTP release R11B) this value does not include reductions performed in current time slices of currently scheduled processes. If an exact value is wanted, use statistics(exact_reductions) [page ??].

`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 `runtime`, except that real time is measured as opposed to runtime or CPU time.

All times are in milliseconds.

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

`erlang:suspend_process(Pid) -> true`

Types:

- Pid = pid()

Suspends the process `Pid`.

Warning:
This BIF is intended for debugging only.

Failure: `badarg` if `Pid` does not exist.

`erlang:system_flag(Flag, Value) -> OldValue`

Types:

- Flag, Value, OldValue – see below

Sets various system properties of the Erlang node. Returns the old value of the flag.

`erlang:system_flag(backtrace_depth, Depth)` Sets the maximum depth of call stack backtraces in the exit reason element of `'EXIT'` tuples.

`erlang:system_flag(fullsweep_after, Number)` `Number` is a 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 0 can help to conserve memory.

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

`erlang:system_flag(min_heap_size, MinHeapSize)` 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/N [page ??] or process_flag/2 [page ??].

`erlang:system_flag(multi_scheduling, BlockState)` `BlockState = block | unblock`

If multi-scheduling is enabled, more than one scheduler thread is used by the emulator. Multi-scheduling can be blocked. When multi-scheduling has been blocked, only one scheduler thread will schedule Erlang processes.

If `BlockState =:= block`, multi-scheduling will be blocked. If `BlockState =:= unblock` and no-one else is blocking multi-scheduling and this process has only blocked one time, multi-scheduling will be unblocked. One process can block multi-scheduling multiple times. If a process has blocked multiple times, it has to unblock exactly as many times as it has blocked before it has released its multi-scheduling block. If a process that has blocked multi-scheduling exits, it will release its blocking of multi-scheduling.

The return values are `disabled`, `blocked`, or `enabled`. The returned value describes the state just after the call to `erlang:system_flag(multi_scheduling, BlockState)` has been made. The return values are described in the documentation of erlang:system_info(multi_scheduling) [page ??].

NOTE: Blocking of multi-scheduling should normally not be needed. If you feel that you need to block multi-scheduling, think through the problem at least a couple of times again. Blocking multi-scheduling should only be used as a last resort since it will most likely be a *very inefficient* way to solve the problem.

See also erlang:system_info(multi_scheduling) [page ??], erlang:system_info(multi_scheduling_blockers) [page ??], and erlang:system_info(schedulers) [page ??].

`erlang:system_flag(trace_control_word, TCW)` Sets the value of the node's trace control word to `TCW`. `TCW` 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:
The `schedulers` option has been removed as of erts version 5.5.3. The number of scheduler threads is determined at emulator boot time, and cannot be changed after that.

`erlang:system_info(Type) -> Res`

Types:

- Type, Res – see below

Returns various information about the current system (emulator) as specified by `Type`:

`allocated_areas` Returns a list of tuples with information about miscellaneous allocated memory areas.

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/0,1 [page ??].

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

Types:

- `Allocator = undefined | elib_malloc | glibc`
- `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 “System Flags Effecting erts_alloc” in [erts_alloc(3)].

`{allocator, Alloc}` Returns information about the specified 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 recognized allocators are listed in [erts_alloc(3)], and after reading this also the returned information should more or less speak for itself. 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.

`check_io` Returns a list containing miscellaneous information regarding the emulators internal I/O checking. Note, the content of the returned list may vary between platforms and over time. The only thing guaranteed is that a list is returned.

`compat_rel` Returns the compatibility mode of the local 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 using the command line flag `+R`, see [erl(1)].

`creation` Returns the creation of the local 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 {`Node, ControllingEntity`}, one entry for each connected remote node. The `Node` 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.

`driver_version` Returns a string containing the erlang driver version used by the runtime system. It will be on the form ["<major ver>.<minor ver>"].

`elib_malloc` If the emulator uses the `elib_malloc` memory allocator, a list of two-element tuples containing status information 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, int()} which is the fullsweep_after garbage collection setting used by default. For more information see garbage_collection described below.

garbage_collection Returns a list describing the default garbage collection settings. A process spawned on the local 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 ??]. spawn_opt/4 [page ??] 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.

kernel_poll Returns true if the emulator uses some kind of kernel-poll implementation; otherwise, false.

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.

modified_timing_level Returns the modified timing level (an integer) if modified timing has been enabled; otherwise, undefined. See the +T command line flag in the documentation of the [erl(1)] command for more information on modified timing.

multi_scheduling Returns disabled, blocked, or enabled. A description of the return values:

disabled The emulator has only one scheduler thread. The emulator does not have SMP support, or have been started with only one scheduler thread.

blocked The emulator has more than one scheduler thread, but all scheduler threads but one have been blocked, i.e., only one scheduler thread will schedule Erlang processes and execute Erlang code.

enabled The emulator has more than one scheduler thread, and no scheduler threads have been blocked, i.e., all available scheduler threads will schedule Erlang processes and execute Erlang code.

See also erlang:system_flag(multi_scheduling, BlockState) [page ??], erlang:system_info(multi_scheduling_blockers) [page ??], and erlang:system_info(schedulers) [page ??].

`multi_scheduling_blockers` Returns a list of PIDs when multi-scheduling is blocked; otherwise, the empty list. The PIDs in the list is PIDs of the processes currently blocking multi-scheduling. A PID will only be present once in the list, even if the corresponding process has blocked multiple times.

See also erlang:system_flag(multi_scheduling, BlockState) [page ??], erlang:system_info(multi_scheduling) [page ??], and erlang:system_info(schedulers) [page ??].

`otp_release` Returns a string containing the OTP release number.

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

`process_limit` Returns the maximum number of concurrently existing processes at the local node as an integer. This limit can be configured at startup by using the command line flag `+P`, see [erl(1)].

`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.

`scheduler_id` Returns the scheduler id (`SchedulerId`) of the scheduler thread that the calling process is executing on. `SchedulerId` is a positive integer; where `1 <= SchedulerId <= erlang:system_info(schedulers)`. See also erlang:system_info(schedulers) [page ??].

`schedulers` Returns the number of scheduler threads used by the emulator. A scheduler thread schedules Erlang processes and Erlang ports, and execute Erlang code and Erlang linked in driver code.

The number of scheduler threads is determined at emulator boot time and cannot be changed after that.

See also erlang:system_info(scheduler_id) [page ??], erlang:system_flag(multi_scheduling, BlockState) [page ??], erlang:system_info(multi_scheduling) [page ??], and and erlang:system_info(multi_scheduling_blockers) [page ??].

`smp_support` Returns `true` if the emulator has been compiled with smp support; otherwise, `false`.

`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 async threads in the async thread pool used for asynchronous driver calls ([driver_async()]) as an integer.

`trace_control_word` Returns the value of the node's trace control word. For more information see documentation of the function `get_tcw` in "Match Specifications in Erlang", [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:**
> The `scheduler` argument has changed name to `scheduler_id`. This in order to avoid mixup with the `schedulers` argument. The `scheduler` argument was introduced in ERTS version 5.5 and renamed in ERTS version 5.5.1.

`erlang:system_monitor() -> MonSettings`

Types:

- MonSettings -> {MonitorPid, Options} | undefined
- MonitorPid = pid()
- Options = [Option]
- Option = {long_gc, Time} | {large_heap, Size} | busy_port | busy_dist_port
- Time = Size = int()

Returns the current system monitoring settings set by erlang:system_monitor/2 [page ??] as `{MonitorPid, Options}`, or `undefined` if there are no settings. The order of the options may be different from the one that was set.

`erlang:system_monitor(undefined | {MonitorPid, Options}) -> MonSettings`

Types:

- MonitorPid, Options, MonSettings – see below

When called with the argument `undefined`, all system performance monitoring settings are cleared.

Calling the function with `{MonitorPid, Options}` as argument, is the same as calling erlang:system_monitor(MonitorPid, Options) [page ??].

Returns the previous system monitor settings just like erlang:system_monitor/0 [page ??].

`erlang:system_monitor(MonitorPid, [Option]) -> MonSettings`

Types:

- MonitorPid = pid()
- Option = {long_gc, Time} | {large_heap, Size} | busy_port | busy_dist_port
- Time = Size = int()
- MonSettings = {OldMonitorPid, [Option]}
- OldMonitorPid = pid()

Sets system performance monitoring options. `MonitorPid` is a local pid that will receive system monitor messages, and the second argument 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 [page ??]).

{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_dist_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 [page ??].

> **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.

Failure: badarg if MonitorPid does not exist.

`term_to_binary(Term) -> ext_binary()`

Types:

- Term = term()

Returns a binary data object which is the result of encoding Term according to the Erlang external term format.

This 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.

See also binary_to_term/1 [page ??].

`term_to_binary(Term, [Option]) -> ext_binary()`

Types:

- Term = term()
- Option = compressed

Returns a binary data object which is the result of encoding Term according to the Erlang external term format.

If the option compressed is provided, the external term format will be compressed. The compressed format is automatically recognized by binary_to_term/1 in R7B and later.

It is also possible to specify a compression level by giving the option {compressed,Level}, where Level is an integer from 0 through 9. 0 means that no compression will be done (it is the same as not giving any compressed option); 1 will take the least time but may not compress as well as the higher levels; 9 will take the most time and may produce a smaller result. Note the "mays" in the preceeding sentence; depending on the input term, level 9 compression may or may not produce a smaller result than level 1 compression.

Currently, compressed gives the same result as {compressed,6}.

The option {minor_version,Version} can be use to control some details of the encoding. This option was introduced in R11B-4. Currently, the allowed values for Version are 0 and 1.

{minor_version,1} forces any floats in the term to be encoded in a more space-efficient and exact way (namely in the 64-bit IEEE format, rather than converted to a textual representation). binary_to_term/1 in R11B-4 and later is able decode the new representation.

{minor_version,0} is currently the default, meaning that floats will be encoded using a textual representation; this option is useful if you want to ensure that releases prior to R11B-4 can decode resulting binary.

See also binary_to_term/1 [page ??].

`throw(Any)`

Types:

- Any = term()

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() -> {Hour, Minute, Second}`

Types:

- Hour = Minute = Second = int()

Returns the current time as {Hour, Minute, Second}.

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

```
> time().
{9,42,44}
```

`tl(List1) -> List2`

Types:

- List1 = List2 = [term()]

Returns the tail of List1, 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 [].

`erlang:trace(PidSpec, How, FlagList) -> int()`

Types:

- PidSpec = pid() | existing | new | all
- How = bool()
- FlagList = [Flag]
- Flag – see below

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 flags (the "message tags" refers to the list of messages following below):

`all` Set all trace flags except {`tracer, Tracer`} and `cpu_timestamp` that are in their nature different than the others.

`send` Trace sending of messages.
Message tags: `send`, `send_to_non_existing_process`.

`'receive'` Trace receiving of messages.
Message tags: `'receive'`.

`procs` Trace process related events.
Message tags: `spawn`, `exit`, `register`, `unregister`, `link`, `unlink`, `getting_linked`, `getting_unlinked`.

`call` Trace certain function calls. Specify which function calls to trace by calling erlang:trace_pattern/3 [page ??].
Message tags: `call`, `return_from`.

`silent` Used in conjunction with the `call` trace flag. The `call`, `return_from` and `return_to` trace messages are inhibited if this flag is set, but if there are match specs they are executed as normal.
Silent mode is inhibited by executing `erlang:trace(_, false, [silent|_])`, or by a match spec executing the {`silent, false`} function.
The `silent` trace flag facilitates setting up a trace on many or even all processes in the system. Then the interesting trace can be activated and deactivated using the {`silent,Bool`} match spec function, giving a high degree of control of which functions with which arguments that triggers the trace.
Message tags: `call`, `return_from`, `return_to`. Or rather, the absence of.

`return_to` Used in conjunction with the `call` trace flag. Trace the actual return from a traced function back to its caller. Only works for functions traced with the `local` option to erlang:trace_pattern/3 [page ??].

The semantics is that a trace message is sent when a call traced function actually returns, that is, 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` Trace scheduling of processes.

Message tags: `in`, `out`.

`garbage_collection` Trace garbage collections of processes.

Message tags: `gc_start`, `gc_end`.

`timestamp` Include 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` Used in conjunction with the `call` trace flag. `{M, F, Arity}` will be specified instead of `{M, F, Args}` in call trace messages.

`set_on_spawn` Makes any process created by a traced process inherit its trace flags, including the `set_on_spawn` flag.

`set_on_first_spawn` Makes the first process created by a traced process inherit its trace flags, excluding the `set_on_first_spawn` flag.

`set_on_link` Makes any process linked by a traced process inherit its trace flags, including the `set_on_link` flag.

`set_on_first_link` Makes the first process linked to by a traced process inherit its trace flags, excluding the `set_on_first_link` flag.

`{tracer, Tracer}` Specify where to send the trace messages. `Tracer` must be the pid of a local process or the port identifier of a local 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. `Pid` is the pid of the traced process in which the traced event has occurred. The third element of the tuple is the message tag.

If the `timestamp` flag is given, the first element of the tuple will be `trace_ts` instead and the timestamp is added last in the tuple.

`{trace, Pid, 'receive', Msg}` When `Pid` receives the message `Msg`.

`{trace, Pid, send, Msg, To}` When `Pid` sends the message `Msg` to the process `To`.

`{trace, Pid, send_to_non_existing_process, Msg, To}` When `Pid` sends the message `Msg` to the non-existing process `To`.

{trace, Pid, call, {M, F, Args}} When Pid calls a traced function. The return values of calls are never supplied, only the call and its arguments.

Note that the trace flag arity can be used to change the contents of this message, so that Arity is specified instead of Args.

{trace, Pid, return_to, {M, F, Args}} When Pid returns *to* the specified function. This trace message is sent if both the call and the return_to flags are set, 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 (that is, the functions match specification matched and {message, false} was not an action).

{trace, Pid, return_from, {M, F, Args}, ReturnValue} When Pid returns *from* the specified function. This trace message is sent if the call flag is set, and the function has a match specification with a return_trace or exception_trace action.

{trace, Pid, exception_from, {M, F, Args}, {Class, Value}} When Pid exits *from* the specified function due to an exception. This trace message is sent if the call flag is set, and the function has a match specification with an exception_trace action.

{trace, Pid, spawn, Pid2, {M, F, Args}} When Pid spawns a new process Pid2 with the specified function call as entry point.

Note that Args 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, RegName} When Pid gets the name RegName registered.

{trace, Pid, unregister, RegName} When Pid gets the name RegName unregistered. Note that this is done automatically when a registered process exits.

{trace, Pid, in, {M, F, Arity} | 0} When Pid is scheduled to run. The process will run in function {M, F, Arity}. On some rare occasions the current function cannot be determined, then the last element Arity is 0.

{trace, Pid, out, {M, F, Arity} | 0} When Pid is scheduled out. The process was running in function {M, F, Arity}. On some rare occasions the current function cannot be determined, then the last element Arity 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 PidSpec 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: If specified arguments are not supported. For example cpu_timestamp is not supported on all platforms.

erlang:trace_delivered(Tracee) -> Ref

Types:

- Tracee = pid() | all
- Ref = reference()

The delivery of trace messages is dislocated on the time-line compared to other events in the system. If you know that the Tracee has passed some specific point in its execution, and you want to know when at least all trace messages corresponding to events up to this point have reached the tracer you can use erlang:trace_delivered(Tracee). A {trace_delivered, Tracee, Ref} message is sent to the caller of erlang:trace_delivered(Tracee) when it is guaranteed that all trace messages have been delivered to the tracer up to the point that the Tracee had reached at the time of the call to erlang:trace_delivered(Tracee).

Note that the trace_delivered message does *not* imply that trace messages have been delivered; instead, it implies that all trace messages that *should* be delivered have been delivered. It is not an error if Tracee isn't, and hasn't been traced by someone, but if this is the case, *no* trace messages will have been delivered when the trace_delivered message arrives.

Note that Tracee has to refer to a process currently, or previously existing on the same node as the caller of erlang:trace_delivered(Tracee) resides on. The special Tracee atom all denotes all processes that currently are traced in the node.

An example: Process A is tracee, port B is tracer, and process C is the port owner of B. C wants to close B when A exits. C can ensure that the trace isn't truncated by calling erlang:trace_delivered(A) when A exits and wait for the {trace_delivered, A, Ref} message before closing B.

Failure: badarg if Tracee does not refer to a process (dead or alive) on the same node as the caller of erlang:trace_delivered(Tracee) resides on.

erlang:trace_info(PidOrFunc, Item) -> Res

Types:

- PidOrFunc = pid() | new | {Module, Function, Arity} | on_load
- Module = Function = atom()
- Arity = int()
- Item, Res – see below

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. Returns undefined if the function does not exist or 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 ??].

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) -> int()

The same as erlang:trace_pattern(MFA, MatchSpec, []) [page ??], retained for backward compatibility.

erlang:trace_pattern(MFA, MatchSpec, FlagList) -> int()

Types:

- MFA, MatchSpec, FlagList – see below

This BIF is used to enable or disable call tracing for exported functions. It must be combined with erlang:trace/3 [page ??] 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 ??] 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 an 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:

`{Module,Function,'_'}` All exported functions of any arity named `Function` in module `Module`.

`{Module,'_','_'}` All exported functions in module `Module`.

`{'_','_','_'}` All exported functions in all loaded modules.

Other combinations, such as `{Module,'_',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 (that is, 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 ??].

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 ??] 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.

`trunc(Number) -> int()`

Types:

- Number = number()

Returns an integer by the truncating `Number`.

```
> trunc(5.5).
5
```

Allowed in guard tests.

`tuple_to_list(Tuple) -> [term()]`

Types:

- Tuple = tuple()

Returns a list which corresponds to `Tuple`. `Tuple` may contain any Erlang terms.

```
> tuple_to_list({share, {'Ericsson_B', 163}}).
[share, {'Ericsson_B', 163}]
```

`erlang:universaltime() -> {Date, Time}`

Types:

- Date = {Year, Month, Day}
- Time = {Hour, Minute, Second}
- Year = Month = Day = Hour = Minute = Second = int()

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({Date1, Time1}) -> {Date2, Time2}`

Types:

- Date1 = Date2 = {Year, Month, Day}
- Time1 = Time2 = {Hour, Minute, Second}
- Year = Month = Day = Hour = Minute = Second = int()

Converts Universal Time Coordinated (UTC) date and time to local date and time, if this is supported by the underlying OS. Otherwise, no conversion is done, and `{Date1, Time1}` is returned.

```
> erlang:universaltime_to_localtime({{1996,11,6},{14,18,43}}).
{{1996,11,7},{15,18,43}}
```

Failure: `badarg` if `Date1` or `Time1` do not denote a valid date or time.

`unlink(Id) -> true`

Types:

- Id = pid() | port()

Removes the link, if there is one, between the calling process and the process or port referred to by `Id`.

Returns `true` and does not fail, even if there is no link to `Id`, or if `Id` does not exist.

Once `unlink(Id)` has returned it is guaranteed that the link between the caller and the entity referred to by `Id` has no effect on the caller in the future (unless the link is setup again). If caller is trapping exits, an `{'EXIT', Id, _}` message due to the link might have been placed in the callers message queue prior to the call, though. Note, the `{'EXIT', Id, _}` message can be the result of the link, but can also be the result of `Id` calling `exit/2`. Therefore, it *may* be appropriate to cleanup the message queue when trapping exits after the call to `unlink(Id)`, as follow:

```
unlink(Id),
receive
    {'EXIT', Id, _} ->
        true
after 0 ->
        true
end
```

> **Note:**
> Prior to OTP release R11B (erts version 5.5) `unlink/1` behaved completely asynchronous, i.e., the link was active until the "unlink signal" reached the linked entity. This had one undesirable effect, though. You could never know when you were guaranteed *not* to be effected by the link.
>
> Current behavior can be viewed as two combined operations: asynchronously send an "unlink signal" to the linked entity and ignore any future results of the link.

`unregister(RegName) -> true`

Types:

- RegName = atom()

Removes the registered name `RegName`, associated with a pid or a port identifier.

```
> unregister(db).
true
```

Users are advised not to unregister system processes.

Failure: `badarg` if `RegName` is not a registered name.

`whereis(RegName) -> pid() | port() | undefined`

Returns the pid or port identifier with the registered name `RegName`. Returns `undefined` if the name is not registered.

```
> whereis(db).
<0.43.0>
```

`erlang:yield() -> true`

Voluntarily let other processes (if any) get a chance to execute. Using `erlang: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, Function, Args) -> term()`

Types:

- Module = Function = atom()
- Args = [term()]
  A (possibly empty) list of arguments `Arg1,..,ArgN`

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

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

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

`undefined_lambda(Module, Fun, Args) -> term()`

Types:

- Module = Function = atom()
- Args = [term()]
  A (possibly empty) list of arguments `Arg1,..,ArgN`

This function is evaluated if a call is made to `Fun(Arg1,.., ArgN)` when the module defining the fun is not loaded. The function is evaluated inside the process making the original call.

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

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

Notes

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

Changes in the code which may seem small can cause a deadlock as unforeseen consequences may occur. The use of `input` is dangerous in this type of code.

error_logger

Erlang Module

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

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

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

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

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

Exports

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

Types:

- Format = string()
- Data = [term()]

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

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

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

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

`error_report(Report) -> ok`

Types:

- Report = [{Tag, Data} | term()] | string() | term()
- Tag = Data = term()

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

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

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

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

`error_report(Type, Report) -> ok`

Types:

- Type = term()
- Report = [{Tag, Data} | term()] | string() | term()
- Tag = Data = term()

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

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

`warning_map() -> Tag`

Types:

- Tag = error | warning | info

Returns the current mapping for warning events. Events sent using warning_msg/1,2 or warning_report/1,2 are tagged as errors (default), warnings or info, depending on the value of the command line flag +W.

```
os$ erl
Erlang (BEAM) emulator version 5.4.8 [hipe] [threads:0] [kernel-poll]

Eshell V5.4.8  (abort with ^G)
1> error_logger:warning_map().
error
2> error_logger:warning_msg("Warnings tagged as: ~p~n", [error]).

=ERROR REPORT==== 11-Aug-2005::15:31:23 ===
Warnings tagged as: error
ok
3>
User switch command
 --> q
os$ erl +W w
Erlang (BEAM) emulator version 5.4.8 [hipe] [threads:0] [kernel-poll]

Eshell V5.4.8  (abort with ^G)
1> error_logger:warning_map().
warning
2> error_logger:warning_msg("Warnings tagged as: ~p~n", [warning]).

=WARNING REPORT==== 11-Aug-2005::15:31:55 ===
Warnings tagged as: warning
ok
```

```
warning_msg(Format) -> ok
warning_msg(Format, Data) -> ok
```

Types:

- Format = string()
- Data = [term()]

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

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

```
warning_report(Report) -> ok
```

Types:

- Report = [{Tag, Data} | term()] | string() | term()
- Tag = Data = term()

Sends a standard warning report event to the error logger. The event is handled by the standard event handler. It is tagged either as an error, warning or info, see warning_map/0 [page ??].

`warning_report(Type, Report) -> ok`

Types:

- Type = term()
- Report = [{Tag, Data} | term()] | string() | term()
- Tag = Data = term()

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

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

Types:

- Format = string()
- Data = [term()]

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

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

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

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

`info_report(Report) -> ok`

Types:

- Report = [{Tag, Data} | term()] | string() | term()
- Tag = Data = term()

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

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

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

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

`info_report(Type, Report) -> ok`

Types:

- Type = term()
- Report = [{Tag, Data} | term()] | string() | term()
- Tag = Data = term()

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

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

`add_report_handler(Handler) -> Result`
`add_report_handler(Handler, Args) -> Result`

Types:

- Handler, Args, Result – see gen_event:add_handler/3

Adds a new event handler to the error logger. The event handler must be implemented as a `gen_event` callback module, see [gen_event(3)].

`Handler` is typically the name of the callback module and `Args` is an optional term (defaults to []) passed to the initialization callback function `Module:init/1`. The function returns `ok` if successful.

The event handler must be able to handle the events [page ??] described below.

`delete_report_handler(Handler) -> Result`

Types:

- Handler, Result – see gen_event:delete_handler/3

Deletes an event handler from the error logger by calling `gen_event:delete_handler(error_logger, Handler, [])`, see [gen_event(3)].

`tty(Flag) -> ok`

Types:

- Flag = bool()

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

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

`logfile(Request) -> ok | Filename | {error, What}`

Types:

- Request = {open, Filename} | close | filename
- Filename = atom() | string()
- What = allready_have_logfile | no_log_file | term()

Enables or disables printout of standard events to a file.

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

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

`Request` is one of:

{`open, Filename`} Opens the log file `Filename`. Returns `ok` if successful, or {`error, allready_have_logfile`} if logging to file is already enabled, or an error tuple if another error occurred. For example, if `Filename` could not be opened.

`close` Closes the current log file. Returns `ok`, or {`error, What`}.

`filename` Returns the name of the log file `Filename`, or {`error, no_log_file`} if logging to file is not enabled.

Events

All event handlers added to the error logger must handle the following events. `Gleader` is the group leader pid of the process which sent the event, and `Pid` is the process which sent the event.

{`error, Gleader, {Pid, Format, Data}`} Generated when `error_msg/1,2` or `format` is called.

{`error_report, Gleader, {Pid, std_error, Report}`} Generated when `error_report/1` is called.

{`error_report, Gleader, {Pid, Type, Report}`} Generated when `error_report/2` is called.

{`warning_msg, Gleader, {Pid, Format, Data}`} Generated when `warning_msg/1,2` is called, provided that warnings are set to be tagged as warnings.

{`warning_report, Gleader, {Pid, std_warning, Report}`} Generated when `warning_report/1` is called, provided that warnings are set to be tagged as warnings.

{`warning_report, Gleader, {Pid, Type, Report}`} Generated when `warning_report/2` is called, provided that warnings are set to be tagged as warnings.

{info_msg, Gleader, {Pid, Format, Data}} Generated when info_msg/1,2 is called.

{info_report, Gleader, {Pid, std_info, Report}} Generated when info_report/1 is called.

{info_report, Gleader, {Pid, Type, Report}} Generated when info_report/2 is called.

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

SEE ALSO

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

file

Erlang Module

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

On operating systems with thread support (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 flag `+A` in [erl(1)].

DATA TYPES

```
iodata() = iolist() | binary()
  iolist() = [char() | binary() | iolist()]

io_device()
  as returned by file:open/2, a process handling IO protocols

name() = string() | atom() | DeepList
  DeepList = [char() | atom() | DeepList]

posix()
  an atom which is named from the Posix error codes used in
  Unix, and in the runtime libraries of most C compilers

time() = {{Year, Month, Day}, {Hour, Minute, Second}}
  Year = Month = Day = Hour = Minute = Second = int()
  Must denote a valid date and time
```

Exports

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

Types:

- Filename = name()
- Gid = int()
- Reason = posix()

Changes group of a file. See write_file_info/2 [page ??].

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

Types:

- Filename = name()
- Uid = int()
- Reason = posix()

Changes owner of a file. See write_file_info/2 [page ??].

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

Types:

- Filename = name()
- Uid = int()
- Gid = int()
- Reason = posix()

Changes owner and group of a file. See write_file_info/2 [page ??].

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

Types:

- Filename = name()
- Mtime = time()
- Reason = posix()

Changes the modification and access times of a file. See write_file_info/2 [page ??].

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

Types:

- Filename = name()
- Mtime = Atime = time()
- Reason = posix()

Changes the modification and last access times of a file. See write_file_info/2 [page ??].

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

Types:

- IoDevice = io_device()
- Reason = posix()

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 [page ??].

`consult(Filename) -> {ok, Terms} | {error, Reason}`

Types:

- Filename = name()
- Terms = [term()]
- Reason = posix() | {Line, Mod, Term}
- Line, Mod, Term – see below

Reads Erlang terms, separated by '.', from `Filename`. Returns one of the following:

{ok, Terms} The file was successfully read.

{error, Posix} An error occurred when opening the file or reading it. See open/2 [page ??] for a list of typical error codes.

{error, {Line, Mod, Term}} An error occurred when interpreting the Erlang terms in the file. Use format_error/1 to convert the three-element tuple to an English description of the error.

Example:

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

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

```
copy(Source, Destination) ->
copy(Source, Destination, ByteCount) -> {ok, BytesCopied} | {error, Reason}
```

Types:

- Source = Destination = io_device() | Filename | {Filename, Modes}
- Filename = name()
- Modes = [Mode] – see open/2
- ByteCount = int() >= 0 | infinity
- BytesCopied = int()

Copies ByteCount bytes from Source to Destination. Source and Destination refer to either filenames or IO devices from e.g. open/2. ByteCount defaults infinity, denoting an infinite number of bytes.

The argument Modes is a list of possible modes, see open/2 [page ??], and defaults to [].

If both Source and Destination refer to filenames, the files are opened with [read, binary] and [write, binary] prepended to their mode lists, respectively, to optimize the copy.

If Source refers to a filename, it is opened with read mode prepended to the mode list before the copy, and closed when done.

If Destination refers to a filename, it is opened with write mode prepended to the mode list before the copy, and closed when done.

Returns {ok, BytesCopied} where BytesCopied is the number of bytes that actually was copied, which may be less than ByteCount if end of file was encountered on the source. If the operation fails, {error, Reason} is returned.

Typical error reasons: As for open/2 if a file had to be opened, and as for read/2 and write/2.

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

Types:

- Dir = name()
- Reason = posix()

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

Typical error reasons are:

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

`eexist` The directory is not empty.

`enoent` The directory does not exist.

`enotdir` A component of `Dir` is not a directory. On some platforms, `enoent` is returned instead.

`einval` Attempt to delete the current directory. On some platforms, `eacces` is returned instead.

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

Types:

- Filename = name()
- Reason = posix()

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

Typical error reasons are:

`enoent` The file does not exist.

`eacces` Missing permission for the file or one of its parents.

`eperm` The file is a directory and the user is not super-user.

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

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

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

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

Types:

- Filename = name()
- Reason = posix() | {Line, Mod, Term}
- Line, Mod, Term – see below

Reads and evaluates Erlang expressions, separated by '.' (or ',', a sequence of expressions is also an expression), from `Filename`. The actual result of the evaluation is not returned; any expression sequence in the file must be there for its side effect. Returns one of the following:

`ok` The file was read and evaluated.

`{error, Posix}` An error occurred when opening the file or reading it. See `open/2` for a list of typical error codes.

`{error, {Line, Mod, Term}}` An error occurred when interpreting the Erlang expressions in the file. Use `format_error/1` to convert the three-element tuple to an English description of the error.

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

Types:

- Filename = name()
- Bindings – see erl_eval(3)
- Reason = posix() | {Line, Mod, Term}
- Line, Mod, Term – see eval/1

The same as `eval/1` but the variable bindings `Bindings` are used in the evaluation. See [erl_eval(3)] about variable bindings.

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

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

`format_error(Reason) -> Chars`

Types:

- Reason = posix() | {Line, Mod, Term}
- Line, Mod, Term – see eval/1
- Chars = [char() | Chars]

Given the error reason returned by any function in this module, returns a descriptive string of the error in English.

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

Types:

- Dir = string()
- Reason = posix()

Returns `{ok, Dir}`, where `Dir` is the current working directory of the file server.

> **Note:**
> In rare circumstances, this function can fail on Unix. It may happen if read permission does not exist for the parent directories of the current directory.

Typical error reasons are:

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

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

Types:

- Drive = string() – see below
- Dir = string()

- Reason = posix()

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

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

Typical error reasons are:

`enotsup` The operating system have no concept of drives.

`eacces` The drive does not exist.

`einval` The format of `Drive` is invalid.

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

Types:

- Dir = name()
- Filenames = [Filename]
- Filename = string()
- Reason = posix()

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

Typical error reasons are:

`eacces` Missing search or write permissions for `Dir` or one of its parent directories.

`enoent` The directory does not exist.

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

Types:

- Dir = name()
- Reason = posix()

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

Typical error reasons are:

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

`eexist` There is already a file or directory named `Dir`.

`enoent` A component of `Dir` does not exist.

`enospc` There is a no space left on the device.

`enotdir` A component of `Dir` is not a directory. On some platforms, `enoent` is returned instead.

`make_link(Existing, New) -> ok | {error, Reason}`

Types:

- Existing = New = name()
- Reason = posix()

Makes a hard link from `Existing` to `New`, on platforms that support links (Unix). This function returns `ok` if the link was successfully created, or `{error, Reason}`. On platforms that do not support links, `{error,enotsup}` is returned.

Typical error reasons:

`eacces` Missing read or write permissions for the parent directories of `Existing` or `New`.

`eexist` `New` already exists.

`enotsup` Hard links are not supported on this platform.

`make_symlink(Name1, Name2) -> ok | {error, Reason}`

Types:

- Name1 = Name2 = name()
- Reason = posix()

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

Typical error reasons:

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

`eexist` `Name2` already exists.

`enotsup` Symbolic links are not supported on this platform.

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

Types:

- Filename = name()
- Modes = [Mode]
- Mode = read | write | append | raw | binary | {delayed_write, Size, Delay} | delayed_write | {read_ahead, Size} | read_ahead | compressed
- Size = Delay = int()
- IoDevice = io_device()
- Reason = posix()

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

`read` The file, which must exist, is opened for reading.

`write` The file is opened for writing. It is created if it does not exist. If the file exists, and if `write` is not combined with `read`, the file will be truncated.

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

`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.

For example, when `delayed_write` is used, after a number of `write/2` calls, `close/1` might return `{error, enospc}` because there was not enough space on the disc for previously written data, and `close/1` should probably be called again since the file is still open.

`delayed_write` The same as `{delayed_write, Size, Delay}` with reasonable default values for `Size` and `Delay`. (Roughly some 64 KBytes, 2 seconds)

`{read_ahead, Size}` This option activates read data buffering. If `read/2` calls are for significantly less than `Size` bytes, read operations towards the operating system are still performed for blocks of `Size` bytes. The extra data is buffered and returned in subsequent `read/2` calls, giving a performance gain since the number of operating system calls is reduced.

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.

Returns:

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

`{error, Reason}` The file could not be opened.

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

> **Note:**
> In previous versions of `file`, modes were given as one of the atoms `read`, `write`, or `read_write` instead of a list. This is still allowed for reasons of backwards compatibility, but should not be used for new code. Also note that `read_write` is not allowed in a mode list.

Typical error reasons:

`enoent` The file does not exist.

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

`eisdir` The named file is not a regular file. It may be a directory, a fifo, or a device.

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

`enospc` There is a no space left on the device (if `write` access was specified).

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

Types:

- Path = [Dir]
- Dir = name()
- Filename = name()
- Terms = [term()]
- FullName = string()
- Reason = posix() | {Line, Mod, Term}
- Line, Mod, Term – see below

Searches the path `Path` (a list of directory names) until the file `Filename` is found. If `Filename` is an absolute filename, `Path` is ignored. Then reads Erlang terms, separated by '.', from the file. Returns one of the following:

`{ok, Terms, FullName}` The file was successfully read. `FullName` is the full name of the file.

`{error, enoent}` The file could not be found in any of the directories in `Path`.

`{error, Posix}` An error occurred when opening the file or reading it. See open/2 [page ??] for a list of typical error codes.

`{error, {Line, Mod, Term}}` An error occurred when interpreting the Erlang terms in the file. Use `format_error/1` to convert the three-element tuple to an English description of the error.

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

Types:

- Path = [Dir]
- Dir = name()
- Filename = name()
- FullName = string()

- Reason = posix() | {Line, Mod, Term}
- Line, Mod, Term – see below

Searches the path `Path` (a list of directory names) until the file `Filename` is found. If `Filename` is an absolute file name, `Path` is ignored. Then reads and evaluates Erlang expressions, separated by '.' (or ',', a sequence of expressions is also an expression), from the file. The actual result of evaluation is not returned; any expression sequence in the file must be there for its side effect. Returns one of the following:

`{ok, FullName}` The file was read and evaluated. `FullName` is the full name of the file.

`{error, enoent}` The file could not be found in any of the directories in `Path`.

`{error, Posix}` An error occurred when opening the file or reading it. See open/2 [page ??] for a list of typical error codes.

`{error, {Line, Mod, Term}}` An error occurred when interpreting the Erlang expressions in the file. Use `format_error/1` to convert the three-element tuple to an English description of the error.

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

Types:

- Path = [Dir]
- Dir = name()
- Filename = name()
- Modes = [Mode] – see open/2
- IoDevice = io_device()
- FullName = string()
- Reason = posix()

Searches the path `Path` (a list of directory names) until the file `Filename` is found. If `Filename` is an absolute file name, `Path` is ignored. Then opens the file in the mode determined by `Modes`. Returns one of the following:

`{ok, IoDevice, FullName}` The file has been opened in the requested mode. `IoDevice` is a reference to the file and `FullName` is the full name of the file.

`{error, enoent}` The file could not be found in any of the directories in `Path`.

`{error, Posix}` The file could not be opened.

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

Types:

- Path = [Dir]
- Dir = name()
- Filename = name()
- Value = term()
- FullName = string()
- Reason = posix() | {Line, Mod, Term}
- Line, Mod, Term – see below

Searches the path `Path` (a list of directory names) until the file `Filename` is found. If `Filename` is an absolute file name, `Path` is ignored. Then reads and evalutates Erlang expressions, separated by '.' (or ',', a sequence of expressions is also an expression), from the file. Returns one of the following:

`{ok, Value, FullName}` The file was read and evaluated. `FullName` is the full name of the file and `Value` the value of the last expression.

`{error, enoent}` The file could not be found in any of the directories in `Path`.

`{error, Posix}` An error occurred when opening the file or reading it. See open/2 [page ??] for a list of typical error codes.

`{error, {Line, Mod, Term}}` An error occurred when interpreting the Erlang expressions in the file. Use `format_error/1` to convert the three-element tuple to an English description of the error.

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

Types:

- Path = [Dir]
- Dir = name()
- Filename = name()
- Bindings – see erl_eval(3)
- Value = term()
- FullName = string()
- Reason = posix() | {Line, Mod, Term}
- Line, Mod, Term – see path_script/2

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) -> string() | undefined`

Types:

- Pid = pid()

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

`{ok, Filename}` If this node's file server is not a slave, the file was opened by this node's file server, (this implies that `Pid` must be a local pid) and the file is not closed. `Filename` is the filename in flat string format.

`undefined` In all other cases.

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

`position(IoDevice, Location) -> {ok, NewPosition} | {error, Reason}`

Types:

- IoDevice = io_device()
- Location = Offset | {bof, Offset} | {cur, Offset} | {eof, Offset} | bof | cur | eof
- Offset = int()
- NewPosition = int()
- Reason = posix()

Sets the position of the file referenced by `IoDevice` to `Location`. Returns {`ok`, `NewPosition`} (as absolute offset) if successful, otherwise {`error`, `Reason`}. `Location` is one of the following:

`Offset` The same as {`bof`, `Offset`}.

{`bof`, `Offset`} Absolute offset.

{`cur`, `Offset`} Offset from the current position.

{`eof`, `Offset`} Offset from the end of file.

`bof` | `cur` | `eof` The same as above with `Offset` 0.

Typical error reasons are:

`einval` Either `Location` was illegal, or it evaluated to a negative offset in the file. Note that if the resulting position is a negative value, the result is an error, and after the call the file position is undefined.

`pread(IoDevice, LocNums) -> {ok, DataL} | {error, Reason}`

Types:

- IoDevice = io_device()
- LocNums = [{Location, Number}]
- Location – see position/2
- Number = int()
- DataL = [Data]
- Data = [char()] | binary() | eof
- Reason = posix()

Performs a sequence of `pread/3` in one operation, which is more efficient than calling them one at a time. Returns {`ok`, `[Data, ...]`} or {`error`, `Reason`}, where each `Data`, the result of the corresponding `pread`, is either a list or a binary depending on the mode of the file, or `eof` if the requested position was beyond end of file.

`pread(IoDevice, Location, Number) -> {ok, Data} | {error, Reason}`

Types:

- IoDevice = io_device()
- Location – see position/2
- Number = int()
- Data = [char()] | binary() | eof
- Reason = posix()

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, LocBytes) -> ok | {error, {N, Reason}}`

Types:

- IoDevice = io_device()
- LocBytes = [{Location, Bytes}]
- Location – see position/2
- Bytes = iodata()
- N = int()
- Reason = posix()

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

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

Types:

- IoDevice = io_device()
- Location – see position/2
- Bytes = iodata()
- Reason = posix()

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) -> {ok, Data} | eof | {error, Reason}`

Types:

- IoDevice = io_device()
- Number = int()
- Data = [char()] | binary()
- Reason = posix()

Reads `Number` bytes from the file referenced 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, Data}` If the file was opened in binary mode, the read bytes are returned in a binary, otherwise in a list. The list or binary will be shorter than the number of bytes requested if end of file was reached.

`eof` Returned if `Number>0` and end of file was reached before anything at all could be read.

`{error, Reason}` An error occurred.

Typical error reasons:

`ebadf` The file is not opened for reading.

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

Types:

- Filename = name()
- Binary = binary()
- Reason = posix()

Returns {ok, Binary}, where Binary is a binary data object that contains the contents of Filename, or {error, Reason} if an error occurs.

Typical error reasons:

enoent The file does not exist.

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

eisdir The named file is a directory.

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

enomem There is not enough memory for the contents of the file.

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

Types:

- Filename = name()
- FileInfo = #file_info{}
- Reason = posix()

Retrieves information about a file. Returns {ok, FileInfo} if successful, otherwise {error, Reason}. FileInfo is a record file_info, defined in the Kernel include file file.hrl. Include the following directive in the module from which the function is called:

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

The record file_info contains the following fields.

size = int() Size of file in bytes.

type = device | directory | regular | other The type of the file.

access = read | write | read_write | none The current system access to the file.

atime = time() The last (local) time the file was read.

mtime = time() The last (local) time the file was written.

ctime = time() The interpretation of this time field depends on the operating system. On Unix, it is the last time the file or the inode was changed. In Windows, it is the create time.

mode = int() The file permissions as the sum of the following bit values:

8#00400 read permission: owner
8#00200 write permission: owner
8#00100 execute permission: owner
8#00040 read permission: group
8#00020 write permission: group
8#00010 execute permission: group
8#00004 read permission: other
8#00002 write permission: other

8#00001 execute permission: other

16#800 set user id on execution

16#400 set group id on execution

On Unix platforms, other bits than those listed above may be set.

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

`major_device = int()` 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 = int()` Only valid for character devices on Unix. In all other cases, this field is zero.

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

`uid = int()` Indicates the owner of the file. Will be zero for non-Unix file systems.

`gid = int()` 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(Name) -> {ok, Filename} | {error, Reason}`

Types:

- Name = name()
- Filename = string()
- Reason = posix()

This function returns `{ok, Filename}` if `Name` refers to a symbolic link or `{error, Reason}` otherwise. On platforms that do not support symbolic links, the return value will be `{error,enotsup}`.

Typical error reasons:

`einval` `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(Name) -> {ok, FileInfo} | {error, Reason}`

Types:

- Name = name()
- FileInfo = #file_info{}, see read_file_info/1
- Reason = posix()

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

If Name is not a symbolic link, this function returns exactly the same result as read_file_info/1. On platforms that do not support symbolic links, this function is always equvivalent to read_file_info/1.

rename(Source, Destination) -> ok | {error, Reason}

Types:

- Source = Destination = name()
- Reason = posix()

Tries to rename the file Source to Destination. It can be used to move files (and directories) between directories, but it is not sufficient to specify the destination only. The destination file name must also be specified. For example, if bar is a normal file and foo and baz are directories, rename("foo/bar", "baz") returns an error, but rename("foo/bar", "baz/bar") succeeds. Returns ok if it is successful.

> **Note:**
> Renaming of open files is not allowed on most platforms (see eacces below).

Typical error reasons:

eacces Missing read or write permissions for the parent directories of Source or Destination. On some platforms, this error is given if either Source or Destination is open.

eexist Destination is not an empty directory. On some platforms, also given when Source and Destination are not of the same type.

einval Source is a root directory, or Destination is a sub-directory of Source.

eisdir Destination is a directory, but Source is not.

enoent Source does not exist.

enotdir Source is a directory, but Destination is not.

exdev Source and Destination are on different file systems.

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

Types:

- Filename = name()
- Value = term()
- Reason = posix() | {Line, Mod, Term}
- Line, Mod, Term – see below

Reads and evaluates Erlang expressions, separated by '.' (or ',', a sequence of expressions is also an expression), from the file. Returns one of the following:

{ok, Value} The file was read and evaluated. Value is the value of the last expression.

{error, Posix} An error occurred when opening the file or reading it. See open/2 [page ??] for a list of typical error codes.

`{error, {Line, Mod, Term}}` An error occurred when interpreting the Erlang expressions in the file. Use `format_error/1` to convert the three-element tuple to an English description of the error.

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

Types:

- Filename = name()
- Bindings – see erl_eval(3)
- Value = term()
- Reason = posix() | {Line, Mod, Term}
- Line, Mod, Term – see below

The same as `script/1` but the variable bindings `Bindings` are used in the evaluation. See [erl_eval(3)] about variable bindings.

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

Types:

- Dir = name()

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

Typical error reasons are:

`enoent` The directory does not exist.

`enotdir` A component of `Dir` is not a directory. On some platforms, `enoent` is returned.

`eacces` Missing permission for the directory or one of its parents.

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

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

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

Types:

- IoDevice = io_device()
- Reason = posix()

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) -> ok | {error, Reason}`

Types:

- IoDevice = io_device()
- Reason = posix()

Truncates the file referenced by `IoDevice` at the current position. Returns `ok` if successful, otherwise {`error, Reason`}.

`write(IoDevice, Bytes) -> ok | {error, Reason}`

Types:

- IoDevice = io_device()
- Bytes = iodata()
- Reason = posix()

Writes `Bytes` to the file referenced by `IoDevice`. This function is the only way to write to a file opened in raw mode (although it works for normally opened files, too). Returns `ok` if successful, and {`error, Reason`} otherwise.

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) -> ok | {error, Reason}`

Types:

- Filename = name()
- Binary = binary()
- Reason = posix()

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

Typical error reasons are:

`enoent` A component of the file name does not exist.

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

`enospc` There is a no space left on the device.

`eacces` Missing permission for writing the file or searching one of the parent directories.

`eisdir` The named file is a directory.

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

Types:

- Filename = name()
- Binary = binary()
- Modes = [Mode] – see open/2
- Reason = posix()

Same as `write_file/2`, but takes a third argument `Modes`, a list of possible modes, see open/2 [page ??]. The mode flags `binary` and `write` are implicit, so they should not be used.

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

Types:

- Filename = name()
- FileInfo = #file_info{} – see also read_file_info/1
- Reason = posix()

Change file information. Returns `ok` if successful, otherwise `{error, Reason}`. `FileInfo` is a record `file_info`, defined in the Kernel include file `file.hrl`. Include the following directive in the module from which the function is called:

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

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

`atime = time()` The last (local) time the file was read.

`mtime = time()` The last (local) time the file was written.

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

`mode = int()` The file permissions as the sum of the following bit values:

8#00400 read permission: owner
8#00200 write permission: owner
8#00100 execute permission: owner
8#00040 read permission: group
8#00020 write permission: group
8#00010 execute permission: group
8#00004 read permission: other
8#00002 write permission: other
8#00001 execute permission: other
16#800 set user id on execution
16#400 set group id on execution

On Unix platforms, other bits than those listed above may be set.

`uid = int()` Indicates the owner of the file. Ignored for non-Unix file systems.

`gid = int()` Gives the group that the owner of the file belongs to. Ignored non-Unix file systems.

Typical error reasons:

`eacces` Missing search permission for one of the parent directories of the file.

`enoent` The file does not exist.

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

POSIX Error Codes

- `eacces` - permission denied
- `eagain` - resource temporarily unavailable
- `ebadf` - bad file number
- `ebusy` - file busy
- `edquot` - disk quota exceeded
- `eexist` - file already exists
- `efault` - bad address in system call argument
- `efbig` - file too large
- `eintr` - interrupted system call
- `einval` - invalid argument
- `eio` - IO error
- `eisdir` - illegal operation on a directory
- `eloop` - too many levels of symbolic links
- `emfile` - too many open files
- `emlink` - too many links
- `enametoolong` - file name too long
- `enfile` - file table overflow
- `enodev` - no such device
- `enoent` - no such file or directory
- `enomem` - not enough memory
- `enospc` - no space left on device
- `enotblk` - block device required
- `enotdir` - not a directory
- `enotsup` - operation not supported
- `enxio` - no such device or address
- `eperm` - not owner
- `epipe` - broken pipe
- `erofs` - read-only file system
- `espipe` - invalid seek
- `esrch` - no such process
- `estale` - stale remote file handle
- `exdev` - cross-domain link

Performance

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

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

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

For increased performance, raw files are recommended. Raw files uses the file system of the node's host machine. For normal files (non-raw), the file server is used to find the files, and if the node is running its file server as slave to another node's, and the other node runs on some other host machine, they may have different file systems. This is seldom a problem, but you have now been warned.

A normal file is really a process so it can be used as an IO device (see `io`). Therefore when data is written to a normal file, the sending of the data to the file process, copies all data that are not binaries. Opening the file in binary mode and writing binaries is therefore recommended. If the file is opened on another node, or if the file server runs as slave to another node's, also binaries are copied.

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

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

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

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

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

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

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

Note:

Trust only your own benchmarks. If the list length in `create_file/2` above is increased, it will run slightly faster, but consume more memory and cause more memory fragmentation. How much this affects your application is something that this simple benchmark can not predict.

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

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

A raw file is really a port. When writing data to a port, it is efficient to write a list of binaries. There is no need to flatten a deep list before writing. On Unix hosts, scatter output, which writes a set of buffers in one operation, is used when possible. In this way `file:write(FD, [Bin1, Bin2 | Bin3])` will write the contents of the binaries without copying the data at all except for perhaps deep down in the operating system kernel.

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

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

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

Warnings

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

SEE ALSO

[filename(3)]

gen_tcp

Erlang Module

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

The following code fragment provides a simple example of a client connecting to a server at port 5678, transferring a binary and closing the connection:

```
client() ->
    SomeHostInNet = "localhost" % to make it runnable on one machine
    {ok, Sock} = gen_tcp:connect(SomeHostInNet, 5678,
                                 [binary, {packet, 0}]),
    ok = gen_tcp:send(Sock, "Some Data"),
    ok = gen_tcp:close(Sock).
```

At the other end a server is listening on port 5678, accepts the connection and receives the binary:

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

do_recv(Sock, Bs) ->
    case gen_tcp:recv(Sock, 0) of
        {ok, B} ->
            do_recv(Sock, [Bs, B]);
        {error, closed} ->
            {ok, list_to_binary(Bs)}
    end.
```

DATA TYPES

```
ip_address()
  see inet(3)

posix()
  see inet(3)

socket()
  as returned by accept/1,2 and connect/3,4
```

Exports

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

Types:

- Address = string() | atom() | ip_address()
- Port = 0..65535
- Options = [Opt]
- Opt – see below
- Timeout = int() | infinity
- Socket = socket()
- Reason = posix()

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

The available options are:

`list` Received `Packet` is delivered as a list.

`binary` Received `Packet` is delivered as a binary.

`{ip, ip_address()}` If the host has several network interfaces, this option specifies which one to use.

`{port, Port}` Specify which local port number to use.

`{fd, int()}` If a socket has somehow been connected without using `gen_tcp`, use this option to pass the file descriptor for it.

`inet6` Set up the socket for IPv6.

`inet` Set up the socket for IPv4.

Opt See inet:setopts/2 [page ??].

Packets can be sent to the returned socket `Socket` using `send/2`. Packets sent from the peer are delivered as messages:

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

If the socket is closed, the following message is delivered:

```
{tcp_closed, Socket}
```

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

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

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

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

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

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

Types:

- Port = 0..65535
- Options = [Opt]
- Opt – see below
- ListenSocket – see below
- Reason = posix()

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

If `Port == 0`, the underlying OS assigns an available port number, use `inet:port/1` to retrieve it.

The available options are:

`list` Received `Packet` is delivered as a list.

`binary` Received `Packet` is delivered as a binary.

`{backlog, B}` `B` is an integer >= 0. The backlog value defaults to 5. The backlog value defines the maximum length that the queue of pending connections may grow to.

`{ip, ip_address()}` If the host has several network interfaces, this option specifies which one to listen on.

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

`inet6` Set up the socket for IPv6.

`inet` Set up the socket for IPv4.

`Opt` See inet:setopts/2 [page ??].

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

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

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

Types:

- ListenSocket – see listen/2
- Timeout = int() | infinity
- Socket = socket()
- Reason = closed | timeout | posix()

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

Returns `{ok, Socket}` if a connection is established, or `{error, closed}` if `ListenSocket` is closed, or `{error, timeout}` if no connection is established within the specified time. May also return a POSIX error value if something else goes wrong, see inet(3) for possible error values.

Packets can be sent to the returned socket `Socket` using `send/2`. Packets sent from the peer are delivered as messages:

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

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

> **Note:**
> It is worth noting that the `accept` call does *not* have to be issued from the socket owner process. Using version 5.5.3 and higher of the emulator, multiple simultaneous accept calls can be issued from different processes, which allows for a pool of acceptor processes handling incoming connections.

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

Types:

- Socket = socket()
- Packet = [char()] | binary()
- Reason = posix()

Sends a packet on a socket.

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

Types:

- Socket = socket()
- Length = int()
- Packet = [char()] | binary()
- Timeout = int() | infinity
- Reason = closed | posix()

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

The `Length` argument is only meaningful when the socket is in `raw` mode and denotes the number of bytes to read. If `Length` = 0, all available bytes are returned. If `Length` > 0, exactly `Length` bytes are returned, or an error; possibly discarding less than `Length` bytes of data when the socket gets closed from the other side.

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

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

Types:

- Socket = socket()
- Pid = pid()

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

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

Types:

- Socket = socket()
- Reason = posix()

Closes a TCP socket.

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

Types:

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

Immediately close a socket in one or two directions.

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

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

gen_udp

Erlang Module

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

DATA TYPES

```
ip_address()
  see inet(3)

posix()
  see inet(3)

socket()
  as returned by open/1,2
```

Exports

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

Types:

- Port = 0..65535
- Options = [Opt]
- Opt – see below
- Socket = socket()
- Reason = posix()

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

The available options are:

`list` Received Packet is delivered as a list.

`binary` Received Packet is delivered as a binary.

{`ip, ip_address()`} If the host has several network interfaces, this option specifies which one to use.

{`fd, int()`} If a socket has somehow been opened without using gen_udp, use this option to pass the file descriptor for it.

`inet6` Set up the socket for IPv6.

`inet` Set up the socket for IPv4.

`Opt` See inet:setopts/2 [page ??].

The returned socket `Socket` is used to send packets from this port with `send/4`. When UDP packets arrive at the opened port, they are delivered as messages:

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

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

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

Default value for the receive buffer option is `{recbuf, 8192}`.

If `Port == 0`, the underlying OS assigns a free UDP port, use `inet:port/1` to retrieve it.

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

Types:

- Socket = socket()
- Address = string() | atom() | ip_address()
- Port = 0..65535
- Packet = [char()] | binary()
- Reason = not_owner | posix()

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

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

Types:

- Socket = socket()
- Length = int()
- Address = ip_address()
- Port = 0..65535
- Packet = [char()] | binary()
- Timeout = int() | infinity
- Reason = not_owner | posix()

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

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

```
controlling_process(Socket, Pid) -> ok
```

Types:

- Socket = socket()
- Pid = pid()

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

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

Types:

- Socket = socket()
- Reason = not_owner | posix()

Closes a UDP socket.

global

Erlang Module

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

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

These services are controlled via the process `global_name_server` which exists on every node. The global name server is started automatically when a node is started. With the term *global* is meant over a system consisting of several Erlang nodes.

The ability to globally register names is a central concept in the programming of distributed Erlang systems. In this module, the equivalent of the `register/2` and `whereis/1` BIFs (for local name registration) are implemented, but for a network of Erlang nodes. A registered name is an alias for a process identifier (pid). The global name server monitors globally registered pids. If a process terminates, the name will also be globally unregistered.

The registered names are stored in replica global name tables on every node. There is no central storage point. Thus, the translation of a name to a pid is fast, as it is always done locally. When any action in taken which results in a change to the global name table, all tables on other nodes are automatically updated.

Global locks have lock identities and are set on a specific resource. For instance, the specified resource could be a pid. When a global lock is set, access to the locked resource is denied for all other resources other than the lock requester.

Both the registration and lock functionalities are atomic. All nodes involved in these actions will have the same view of the information.

The global name server also performs the critical task of continuously monitoring changes in node configuration: if a node which runs a globally registered process goes down, the name will be globally unregistered. To this end the global name server subscribes to `nodeup` and `nodedown` messages sent from the `net_kernel` module. Relevant Kernel application variables in this context are `net_setuptime`, `net_ticktime`, and `dist_auto_connect`. See also kernel(6) [page ??].

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

If the global name server fails to connect nodes (`N1` and `N3` in the example above) a warning event is sent to the error logger. The presence of such an event does not exclude the possibility that the nodes will later connect–one can for example try the

command `rpc:call(N1, net_adm, ping, [N2])` in the Erlang shell–but it indicates some kind of problem with the network.

> **Note:**
> If the fully connected network is not set up properly, the first thing to try is to increase the value of `net_setuptime`.

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`

Types:

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

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

`random_exit_name(Name, Pid1, Pid2) -> Pid1 | Pid2`

Types:

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

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

`random_notify_name(Name, Pid1, Pid2) -> Pid1 | Pid2`

Types:

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

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

```
register_name(Name, Pid)
register_name(Name, Pid, Resolve) -> yes | no
```

Types:

- Name = term()
- Pid = pid()
- Resolve = fun() or {Module, Function} where
- Resolve(Name, Pid, Pid2) -> Pid | Pid2 | none

Globally associates the name `Name` with a pid, that is, Globally notifies all nodes of a new global name in a network of Erlang nodes.

When new nodes are added to the network, they are informed of the globally registered names that already exist. The network is also informed of any global names in newly connected nodes. If any name clashes are discovered, the `Resolve` function is called. Its purpose is to decide which pid is correct. If the function crashes, or returns anything other than one of the pids, the name is unregistered. This function is called once for each name clash.

There are three pre-defined resolve functions: `random_exit_name/3`, `random_notify_name/3`, and `notify_all_name/3`. If no `Resolve` function is defined, `random_exit_name` is used. This means that one of the two registered processes will be selected as correct while the other is killed.

This function is completely synchronous. This means that when this function returns, the name is either registered on all nodes or none.

The function returns `yes` if successful, `no` if it fails. For example, `no` is returned if an attempt is made to register an already registered process or to register a process with a name that is already in use.

> **Note:**
> Releases up to and including OTP R10 did not check if the process was already registered. As a consequence the global name table could become inconsistent. The old (buggy) behavior can be chosen by giving the Kernel application variable `global_multi_name_action` the value `allow`.

If a process with a registered name dies, or the node goes down, the name is unregistered on all nodes.

```
registered_names() -> [Name]
```

Types:

- Name = term()

Returns a lists of all globally registered names.

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

Types:

- Name = term()
- Pid = pid()

- Resolve = fun() or {Module, Function} where
- Resolve(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/2,3`.

`send(Name, Msg) -> Pid`

Types:

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

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

Failure: If `Name` is not a globally registered name, the calling function will exit with reason {`badarg, {Name, Msg}`}.

`set_lock(Id)`

`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.

The global name server keeps track of all processes sharing the same lock, that is, if two processes set the same lock, both processes must delete the lock.

This function does not address the problem of a deadlock. A deadlock can never occur as long as processes only lock one resource at a time. But if some processes try to lock two or more resources, a deadlock may occur. It is up to the application to detect and rectify a deadlock.

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

```
sync() -> 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`). If this succeeds, `Fun()` is evaluated and the result `Res` is returned. Returns `aborted` if the lock attempt failed. If `Retries` is set to `infinity`, the transaction will not abort.

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

```
unregister_name(Name) -> void()
```

Types:

- Name = term()

Removes the globally registered name `Name` from the network of Erlang nodes.

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

Types:

- Name = term()

Returns the pid with the globally registered name `Name`. Returns `undefined` if the name is not globally registered.

See Also

global_group(3) [page ??], net_kernel(3) [page ??],

global_group

Erlang Module

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

The main advantage of dividing systems to global groups is that the background load decreases while the number of nodes to be updated is reduced when manipulating globally registered names.

The Kernel configuration parameter `global_groups` defines the global groups (see also kernel(6) [page ??], config(4) [page ??]:

```
{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 == normal`.

A node started with the command line flag `-hidden`, see [erl(1)], is said to be a *hidden* node. A hidden node will establish hidden connections to nodes not part of the same global group, but normal (visible) connections to nodes part of the same global group.

A global group defined with `PublishType == hidden`, is said to be a hidden global group. All nodes in a hidden global group are hidden nodes, regardless if they are started with the `-hidden` command line flag or not.

For the processes and nodes to run smoothly using the global group functiontionality, the following criteria must be met:

- An instance of the global group server, `global_group`, must be running on each node. The processes are automatically started and synchronized when a node is started.
- All involved nodes must agree on the global group definition, or the behavior of the system is undefined.
- *All* nodes in the system should belong to exactly one global group.

In the following description, a *group node* is a node belonging to the same global group as the local node.

Exports

`global_groups() -> {GroupName, GroupNames} | undefined`

Types:

- GroupName = atom()
- GroupNames = [GroupName]

Returns a tuple containing the name of the global group the local node belongs to, and the list of all other known group names. Returns `undefined` if no global groups are defined.

`info() -> [{Item, Info}]`

Types:

- Item, Info – see below

Returns a list containing information about the global groups. Each element of the list is a tuple. The order of the tuples is not defined.

`{state, State}` If the local node is part of a global group, `State == synced`. If no global groups are defined, `State == no_conf`.

`{own_group_name, GroupName}` The name (atom) of the group that the local node belongs to.

`{own_group_nodes, Nodes}` A list of node names (atoms), the group nodes.

`{synced_nodes, Nodes}` A list of node names, the group nodes currently synchronized with the local node.

`{sync_error, Nodes}` A list of node names, the group nodes with which the local node has failed to synchronize.

`{no_contact, Nodes}` A list of node names, the group nodes to which there are currently no connections.

`{other_groups, Groups}` `Groups` is a list of tuples `{GroupName, Nodes}`, specifying the name and nodes of the other global groups.

`{monitoring, Pids}` A list of pids, specifying the processes which have subscribed to `nodeup` and `nodedown` messages.

`monitor_nodes(Flag) -> ok`

Types:

- Flag = bool()

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

A process which has subscribed will receive the messages `{nodeup, Node}` and `{nodedown, Node}` when a group node connects or disconnects, respectively.

`own_nodes() -> Nodes`

Types:

- Nodes = [Node]
- Node = node()

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

```
registered_names(Where) -> Names
```

Types:

- Where = {node, Node} | {group, GroupName}
- Node = node()
- GroupName = atom()
- Names = [Name]
- Name = atom()

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

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

Types:

- Where = {node, Node} | {group, GroupName}
- Node = node()
- GroupName = atom()
- Name = atom()
- Msg = term()

Searches for `Name`, globally registered on the specified node or in the specified global group, or – if the `Where` argument is not provided – in any global group. The global groups are searched in the order in which they appear in the value of the `global_groups` configuration parameter.

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

```
sync() -> ok
```

Synchronizes the group nodes, that is, the global name servers on the group nodes. Also check the names globally registered in the current global group and unregisters them on any known node not part of the group.

If synchronization is not possible, an error report is sent to the error logger (see also `error_logger(3)`).

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

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

Types:

- Where = {node, Node} | {group, GroupName}
- Node = node()
- GroupName = atom()
- Name = atom()

Searches for `Name`, globally registered on the specified node or in the specified global group, or – if the `Where` argument is not provided – in any global group. The global groups are searched in the order in which they appear in the value of the `global_groups` configuration parameter.

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

NOTE

In the situation where a node has lost its connections to other nodes in its global group, but has connections to nodes in other global groups, a request from another global group may produce an incorrect or misleading result. For example, the isolated node may not have accurate information about registered names in its global group.

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

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

SEE ALSO

[erl(1)], global(3) [page ??]

heart

Erlang Module

This modules contains the interface to the heart process. heart sends periodic heartbeats to an external port program, which is also named heart. The purpose of the heart port program is to check that the Erlang runtime system it is supervising is still running. If the port program has not received any heartbeats within HEART_BEAT_TIMEOUT seconds (default is 60 seconds), the system can be rebooted. Also, if the system is equipped with a hardware watchdog timer and is running Solaris, the watchdog can be used to supervise the entire system.

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

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

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

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

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

The HEART_BEAT_TIMEOUT and HEART_BEAT_BOOT_DELAY environment variables can be used to configure the heart timeouts, they can be set in the operating system shell before Erlang is started or be specified at the command line:

```
% erl -heart -env HEART_BEAT_TIMEOUT 30 ...
```

The value (in seconds) must be in the range 10 < X <= 65535.

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

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

Exports

`set_cmd(Cmd) -> ok | {error, {bad_cmd, Cmd}}`

Types:

- Cmd = string()

Sets a temporary reboot command. This command is used if a `HEART_COMMAND` other than the one specified with the environment variable should be used in order to reboot the system. The new Erlang runtime system will (if it misbehaves) use the environment variable `HEART_COMMAND` to reboot.

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

`clear_cmd() -> ok`

Clears the temporary boot command. If the system terminates, the normal `HEART_COMMAND` is used to reboot.

`get_cmd() -> {ok, Cmd}`

Types:

- Cmd = string()

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

inet

Erlang Module

Provides access to TCP/IP protocols.

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

Two Kernel configuration parameters affect the behaviour of all sockets opened on an Erlang node: `inet_default_connect_options` can contain a list of default options used for all sockets returned when doing `connect`, and `inet_default_listen_options` can contain a list of default options used when issuing a `listen` call. When `accept` is issued, the values of the listensocket options are inherited, why no such application variable is needed for `accept`.

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

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

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

DATA TYPES

```
#hostent{h_addr_list = [ip_address()]  % list of addresses for this host
         h_addrtype  = inet | inet6
         h_aliases = [hostname()]      % list of aliases
         h_length = int()              % length of address in bytes
         h_name = hostname()           % official name for host
  The record is defined in the Kernel include file "inet.hrl"
  Add the following directive to the module:
    -include_lib("kernel/include/inet.hrl").

hostname() = atom() | string()

ip_address() = {N1,N2,N3,N4}              % IPv4
             | {K1,K2,K3,K4,K5,K6,K7,K8}  % IPv6
  Ni = 0..255
  Ki = 0..65535

posix()
```

```
  an atom which is named from the Posix error codes used in
  Unix, and in the runtime libraries of most C compilers
```

```
socket()
  see gen_tcp(3), gen_udp(3)
```

Addresses as inputs to functions can be either a string or a tuple. For instance, the IP address 150.236.20.73 can be passed to `gethostbyaddr/1` either as the string "150.236.20.73" or as the tuple {150, 236, 20, 73}.

IPv4 address examples:

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

IPv6 address examples:

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

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

```
1> inet_parse:address("192.168.42.2").
{ok,{192,168,42,2}}
2> inet_parse:address("FFFF::192.168.42.2").
{ok,{65535,0,0,0,0,0,49320,10754}}
```

Exports

`close(Socket) -> ok`

Types:

- Socket = socket()

Closes a socket of any type.

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

Types:

- Par, Val – see below

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

`format_error(Posix) -> string()`

Types:

- Posix = posix()

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

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

Types:

- Host = ip_address() | string() | atom()
- Family = inet | inet6
- Address = ip_address()
- posix() = term()

Returns the IP-address for `Host` as a tuple of integers. `Host` can be an IP-address, a single hostname or a fully qualified hostname.

`getaddrs(Host, Family) -> {ok, Addresses} | {error, posix()}`

Types:

- Host = ip_address() | string() | atom()
- Addresses = [ip_address()]
- Family = inet | inet6

Returns a list of all IP-addresses for `Host`. `Host` can be an IP-adress, a single hostname or a fully qualified hostname.

`gethostbyaddr(Address) -> {ok, Hostent} | {error, posix()}`

Types:

- Address = string() | ip_address()
- Hostent = #hostent{}

Returns a `hostent` record given an address.

`gethostbyname(Name) -> {ok, Hostent} | {error, posix()}`

Types:

- Hostname = hostname()
- Hostent = #hostent{}

Returns a `hostent` record given a hostname.

`gethostbyname(Name, Family) -> {ok, Hostent} | {error, posix()}`

Types:

- Hostname = hostname()
- Family = inet | inet6
- Hostent = #hostent{}

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

`gethostname() -> {ok, Hostname} | {error, posix()}`

Types:

- Hostname = string()

Returns the local hostname. Will never fail.

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

Types:

- Socket = term()
- Options = [Opt | RawOptReq]
- Opt = atom()
- RawOptReq = {raw, Protocol, OptionNum, ValueSpec}
- Protocol = int()
- OptionNum = int()
- ValueSpec = ValueSize | ValueBin
- ValueSize = int()
- ValueBin = binary()
- OptionValues = [{Opt, Val} | {raw, Protocol, OptionNum, ValueBin}]

Gets one or more options for a socket. See [inet:setopts/2] for a list of available options.

The number of elements in the returned `OptionValues` list does not necessarily correspond to the number of options asked for. If the operating system fails to support an option, it is simply left out in the returned list. An error tuple is only returned when getting options for the socket is impossible (i.e. the socket is closed or the buffer size in a raw request is too large). This behavior is kept for backward compatibility reasons.

A `RawOptReq` can be used to get information about socket options not (explicitly) supported by the emulator. The use of raw socket options makes the code non portable, but allows the Erlang programmer to take advantage of unusual features present on the current platform.

The `RawOptReq` consists of the tag `raw` followed by the protocol level, the option number and either a binary or the size, in bytes, of the buffer in which the option value is to be stored. A binary should be used when the underlying `getsockopt` requires *input* in the argument field, in which case the size of the binary should correspond to the required buffer size of the return value. The supplied values in a `RawOptReq` correspond to the second, third and fourth/fifth parameters to the `getsockopt` call in the C socket API. The value stored in the buffer is returned as a binary `ValueBin` where all values are coded in the native endianess.

Asking for and inspecting raw socket options require low level information about the current operating system and TCP stack.

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

```
get_tcpi_sacked(Sock) ->
    {ok,[{raw,_,_,Info}]} = inet:getopts(Sock,[{raw,6,11,92}]),
    <<_:28/binary,TcpiSacked:32/native,_/binary>> = Info,
    TcpiSacked.
```

Preferably, you would check the machine type, the OS and the kernel version prior to executing anything similar to the code above.

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

Types:

- Socket = socket()
- Address = ip_address()
- Port = int()

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

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

Types:

- Socket = socket()
- Port = int()

Returns the local port number for a socket.

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

Types:

- Socket = socket()
- Address = ip_address()
- Port = int()

Returns the local address and port number for a socket.

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

Types:

- Socket = term()
- Options = [{Opt, Val} | {raw, Protocol, Option, ValueBin}]
- Protocol = int()
- OptionNum = int()
- ValueBin = binary()
- Opt, Val – see below

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

{active, Boolean} If the value is true, which is the default, everything received from the socket will be sent as messages to the receiving process. If the value is false (passive mode), the process must explicitly receive incoming data by calling gen_tcp:recv/2,3 or gen_udp:recv/2,3 (depending on the type of socket).

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

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} (UDP sockets) Enable/disable permission to send broadcasts.

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

{dontroute, Boolean} Enable/disable routing bypass for outgoing messages.

{exit_on_close, Boolean} By default this option is set to true.

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

{header, Size} This option is only meaningful if the binary option was specified when the socket was created. If the header option is specified, the first Size number bytes of data received from the socket will be elements of a list, and the rest of the data will be a binary given as the tail of the same list. If for example Size == 2, the data received will match [Byte1,Byte2|Binary].

{keepalive, Boolean} (TCP/IP sockets) Enables/disables periodic transmission on a connected socket, when no other data is being exchanged. If the other end does not respond, the connection is considered broken and an error message will be sent to the controlling process. Default disabled.

{nodelay, Boolean} (TCP/IP sockets) If Boolean == true, the TCP_NODELAY option is turned on for the socket, which means that even small amounts of data will be sent immediately.

{packet, PacketType} (TCP/IP sockets) Defines the type of packets to use for a socket. The following values are valid:

raw | 0 No packaging is done.

1 | 2 | 4 Packets consist of a header specifying the number of bytes in the packet, followed by that number of bytes. The length of header can be one, two, or four bytes; 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/2,3` returns one complete packet. The header is *not* stripped off.
The meanings of the packet types are as follows:
`asn1` - ASN.1 BER,
`sunrm` - Sun's RPC encoding,
`cdr` - CORBA (GIOP 1.1),
`fcgi` - Fast CGI,
`tpkt` - TPKT format [RFC1006],
`line` - Line mode, a packet is a line terminated with newline, lines longer than the receive buffer are truncated.

{`packet_size, Integer`} (TCP/IP sockets) Sets the max allowed length of the packet body. If the packet header indicates that the length of the packet is longer than the max allowed length, the packet is considered invalid. The same happens if the packet header is too big for the socket receive buffer.

{`read_packets, Integer`} (UDP sockets) Sets the max number of UDP packets to read without intervention from the socket when data is available. When this many packets have been read and delivered to the destination process, new packets are not read until a new notification of available data has arrived. The default is 5, and if this parameter is set too high the system can become unresponsive due to UDP packet flooding.

{`recbuf, Integer`} Gives the size of the receive buffer to use for the socket.

{`reuseaddr, Boolean`} Allows or disallows local reuse of port numbers. By default, reuse is disallowed.

{`sndbuf, Integer`} Gives the size of the send buffer to use for the socket.

{`priority, Integer`} Sets the SO_PRIORITY socket level option on platforms where this is implemented. The behaviour and allowed range varies on different systems. The option is ignored on platforms where the option is not implemented. Use with caution.

{`tos, Integer`} Sets IP_TOS IP level options on platforms where this is implemented. The behaviour and allowed range varies on different systems. The option is ignored on platforms where the option is not implemented. Use with caution.

In addition to the options mentioned above, *raw* option specifications can be used. The raw options are specified as a tuple of arity four, beginning with the tag `raw`, followed by the protocol level, the option number and the actual option value specified as a binary. This corresponds to the second, third and fourth argument to the `setsockopt` call in the C socket API. The option value needs to be coded in the native endianess of the platform and, if a structure is required, needs to follow the struct alignment conventions on the specific platform.

Using raw socket options require detailed knowledge about the current operating system and TCP stack.

As an example of the usage of raw options, consider a Linux system where you want to set the `TCP_LINGER2` option on the `IPPROTO_TCP` protocol level in the stack. You know that on this particular system it defaults to 60 (seconds), but you would like to lower it to 30 for a particular socket. The `TCP_LINGER2` option is not explicitly supported by inet, but you know that the protocol level translates to the number 6, the option number to the number 8 and the value is to be given as a 32 bit integer. You can use this line of code to set the option for the socket named `Sock`:

```
inet:setopts(Sock,[{raw,6,8,<<30:32/native>>}]),
```

As many options are silently discarded by the stack if they are given out of range, it could be a good idea to check that a raw option really got accepted. This code places the value in the variable TcpLinger2:

```
{ok,[{raw,6,8,<<TcpLinger2:32/native>>}]}=inet:getopts(Sock,[{raw,6,8,4}]),
```

Code such as the examples above is inherently non portable, even different versions of the same OS on the same platform may respond differently to this kind of option manipulation. Use with care.

Note that the default options for TCP/IP sockets can be changed with the Kernel configuration parameters mentioned in the beginning of this document.

POSIX Error Codes

- `e2big` - argument list too long
- `eacces` - permission denied
- `eaddrinuse` - address already in use
- `eaddrnotavail` - cannot assign requested address
- `eadv` - advertise error
- `eafnosupport` - address family not supported by protocol family
- `eagain` - resource temporarily unavailable
- `ealign` - EALIGN
- `ealready` - operation already in progress
- `ebade` - bad exchange descriptor
- `ebadf` - bad file number
- `ebadfd` - file descriptor in bad state
- `ebadmsg` - not a data message
- `ebadr` - bad request descriptor
- `ebadrpc` - RPC structure is bad
- `ebadrqc` - bad request code
- `ebadslt` - invalid slot
- `ebfont` - bad font file format
- `ebusy` - file busy
- `echild` - no children
- `echrng` - channel number out of range
- `ecomm` - communication error on send
- `econnaborted` - software caused connection abort
- `econnrefused` - connection refused
- `econnreset` - connection reset by peer
- `edeadlk` - resource deadlock avoided
- `edeadlock` - resource deadlock avoided
- `edestaddrreq` - destination address required
- `edirty` - mounting a dirty fs w/o force

- `edom` - math argument out of range
- `edotdot` - cross mount point
- `edquot` - disk quota exceeded
- `eduppkg` - duplicate package name
- `eexist` - file already exists
- `efault` - bad address in system call argument
- `efbig` - file too large
- `ehostdown` - host is down
- `ehostunreach` - host is unreachable
- `eidrm` - identifier removed
- `einit` - initialization error
- `einprogress` - operation now in progress
- `eintr` - interrupted system call
- `einval` - invalid argument
- `eio` - I/O error
- `eisconn` - socket is already connected
- `eisdir` - illegal operation on a directory
- `eisnam` - is a named file
- `el2hlt` - level 2 halted
- `el2nsync` - level 2 not synchronized
- `el3hlt` - level 3 halted
- `el3rst` - level 3 reset
- `elbin` - ELBIN
- `elibacc` - cannot access a needed shared library
- `elibbad` - accessing a corrupted shared library
- `elibexec` - cannot exec a shared library directly
- `elibmax` - attempting to link in more shared libraries than system limit
- `elibscn` - .lib section in a.out corrupted
- `elnrng` - link number out of range
- `eloop` - too many levels of symbolic links
- `emfile` - too many open files
- `emlink` - too many links
- `emsgsize` - message too long
- `emultihop` - multihop attempted
- `enametoolong` - file name too long
- `enavail` - not available
- `enet` - ENET
- `enetdown` - network is down
- `enetreset` - network dropped connection on reset
- `enetunreach` - network is unreachable
- `enfile` - file table overflow

- `enoano` - anode table overflow
- `enobufs` - no buffer space available
- `enocsi` - no CSI structure available
- `enodata` - no data available
- `enodev` - no such device
- `enoent` - no such file or directory
- `enoexec` - exec format error
- `enolck` - no locks available
- `enolink` - link has be severed
- `enomem` - not enough memory
- `enomsg` - no message of desired type
- `enonet` - machine is not on the network
- `enopkg` - package not installed
- `enoprotoopt` - bad 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

The `init` module is pre-loaded and contains the code for the `init` system process which coordinates the start-up of the system. The first function evaluated at start-up is `boot(BootArgs)`, where `BootArgs` is a list of command line arguments supplied to the Erlang runtime system from the local operating system. See [erl(1)].

`init` reads the boot script which contains instructions on how to initiate the system. See [script(4)] for more information about boot scripts.

`init` also contains functions to restart, reboot, and stop the system.

Exports

`boot(BootArgs) -> void()`

Types:

- BootArgs = [binary()]

Starts the Erlang runtime system. This function is called when the emulator is started and coordinates system start-up.

`BootArgs` are all command line arguments except the emulator flags, that is, flags and plain arguments. See [erl(1)].

`init` itself interprets some of the flags, see Command Line Flags [page ??] below. The remaining flags ("user flags") and plain arguments are passed to the `init` loop and can be retrieved by calling `get_arguments/0` and `get_plain_arguments/0`, respectively.

`get_args() -> [Arg]`

Types:

- Arg = atom()

Returns any plain command line arguments as a list of atoms (possibly empty). It is recommended that `get_plain_arguments/1` is used instead, because of the limited length of atoms.

`get_argument(Flag) -> {ok, Arg} | error`

Types:

- Flag = atom()
- Arg = [Values]
- Values = [string()]

Returns all values associated with the command line user flag `Flag`. If `Flag` is provided several times, each `Values` is returned in preserved order.

```
% erl -a b c -a d
...
1> init:get_argument(a).
{ok,[["b","c"],["d"]]}
```

There are also a number of flags, which are defined automatically and can be retrieved using this function:

`root` The installation directory of Erlang/OTP, `$ROOT`.

```
2> init:get_argument(root).
{ok,[["/usr/local/otp/releases/otp_beam_solaris8_r10b_patched"]]}
```

`progname` The name of the program which started Erlang.

```
3> init:get_argument(progname).
{ok,[["erl"]]}
```

`home` The home directory.

```
4> init:get_argument(home).
{ok,[["/home/harry"]]}
```

Returns `error` if there is no value associated with `Flag`.

`get_arguments() -> Flags`

Types:

- Flags = [{Flag, Values}]
- Flag = atom()
- Values = [string()]

Returns all command line flags, as well as the system defined flags, see `get_argument/1`.

`get_plain_arguments() -> [Arg]`

Types:

- Arg = string()

Returns any plain command line arguments as a list of strings (possibly empty).

`get_status() -> {InternalStatus, ProvidedStatus}`

Types:

- InternalStatus = starting | started | stopping
- ProvidedStatus = term()

The current status of the `init` process can be inspected. During system startup (initialization), `InternalStatus` is `starting`, and `ProvidedStatus` indicates how far the boot script has been interpreted. Each `{progress, Info}` term interpreted in the boot script affects `ProvidedStatus`, that is, `ProvidedStatus` gets the value of `Info`.

`reboot() -> void()`

All applications are taken down smoothly, all code is unloaded, and all ports are closed before the system terminates. If the `-heart` command line flag was given, the `heart` program will try to reboot the system. Refer to `heart(3)` for more information.

To limit the shutdown time, the time `init` is allowed to spend taking down applications, the `-shutdown_time` command line flag should be used.

`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.

`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.

`stop() -> void()`

All applications are taken down smoothly, all code is unloaded, and all ports are closed before the system terminates. If the `-heart` command line flag was given, the `heart` program is terminated before the Erlang node terminates. Refer to `heart(3)` for more information.

To limit the shutdown time, the time `init` is allowed to spend taking down applications, the `-shutdown_time` command line flag should be used.

Command Line Flags

The `init` module interprets the following command line flags:

`--` Everything following `--` up to the next flag is considered plain arguments and can be retrieved using `get_plain_arguments/0`.

`-eval Expr` Scans, parses and evaluates an arbitrary expression `Expr` during system initialization. If any of these steps fail (syntax error, parse error or exception during evaluation), Erlang stops with an error message. Here is 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 halt
BF
```

If multiple `-eval` expressions are specified, they are 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 does not terminate, blocks the system initialization process.

`-extra` Everything following `-extra` is considered plain arguments and can be retrieved using `get_plain_arguments/0`.

`-run Mod [Func [Arg1, Arg2, ...]]` Evaluates the specified function call during system initialization. `Func` defaults to `start`. If no arguments are provided, the function is assumed to be of arity 0. Otherwise it is assumed to be of arity 1, taking the list `[Arg1,Arg2,...]` as argument. All arguments are passed as strings. If an exception is raised, Erlang stops with an error message.

Example:

```
% erl -run foo -run foo bar -run foo bar baz 1 2
```

This starts the Erlang runtime system and evaluates the following functions:

```
foo:start()
foo:bar()
foo:bar(["baz", "1", "2"]).
```

The functions are executed sequentially in an initialization process, which then terminates normally and passes control to the user. This means that a `-run` call which does not return will block further processing; to avoid this, use some variant of `spawn` in such cases.

`-s Mod [Func [Arg1, Arg2, ...]]` Evaluates the specified function call during system initialization. `Func` defaults to `start`. If no arguments are provided, the function is assumed to be of arity 0. Otherwise it is assumed to be of arity 1, taking the list `[Arg1,Arg2,...]` as argument. All arguments are passed as atoms. If an exception is raised, Erlang stops with an error message.

Example:

```
% erl -s foo -s foo bar -s foo bar baz 1 2
```

This starts the Erlang runtime system and evaluates the following functions:

```
foo:start()
foo:bar()
foo:bar([baz, '1', '2']).
```

The functions are executed sequentially in an initialization process, which then terminates normally and passes control to the user. This means that a `-s` call which does not return will block further processing; to avoid this, use some variant of `spawn` in such cases.

Due to the limited length of atoms, it is recommended that `-run` be used instead.

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 ??], heart(3) [page ??]

net_adm

Erlang Module

This module contains various network utility functions.

Exports

`dns_hostname(Host) -> {ok, Name} | {error, Host}`

Types:

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

Returns the official name of `Host`, or `{error, Host}` if no such name is found. See also `inet(3)`.

`host_file() -> Hosts | {error, Reason}`

Types:

- Hosts = [Host]
- Host = atom()
- Reason = term()

Reads the `.hosts.erlang` file, see the section *Files* below. Returns the hosts in this file as a list, or returns `{error, Reason}` if the file could not be read. See `file(3)` for possible values of `Reason`.

`localhost() -> Name`

Types:

- Name = string()

Returns the name of the local host. If Erlang was started with the `-name` command line flag, `Name` is the fully qualified name.

`names() -> {ok, [{Name, Port}]} | {error, Reason}`
`names(Host) -> {ok, [{Name, Port}]} | {error, Reason}`

Types:

- Name = string()
- Port = int()
- Reason = address | term()

Similar to `epmd -names`, see `epmd(1)`. `Host` defaults to the local host. Returns the names and associated port numbers of the Erlang nodes that `epmd` at the specified host has registered.

Returns `{error, address}` if `epmd` is not running. See `inet(3)` for other possible values of `Reason`.

```
(arne@dunn)1> net_adm:names().
{ok,[{"arne",40262}]}
```

```
ping(Node) -> pong | pang
```

Types:

- Node = node()

Tries to set up a connection to `Node`. Returns `pang` if it fails, or `pong` if it is successful.

```
world() -> [node()]
world(Arg) -> [node()]
```

Types:

- Arg = silent | verbose

This function calls `names(Host)` for all hosts which are specified in the Erlang host file `.hosts.erlang`, collects the replies and then evaluates `ping(Node)` on all those nodes. Returns the list of all found nodes, regardless of the return value of `ping(Node)`.

`Arg` defaults to `silent`. If `Arg == verbose`, the function writes information about which nodes it is pinging to stdout.

This function can be useful when a node is started, and the names of the other nodes in the network are not initially known.

Failure: `{error, Reason}` if `host_file()` returns `{error, Reason}`.

```
world_list(Hosts) -> [node()]
world_list(Hosts, Arg) -> [node()]
```

Types:

- Hosts = [Host]
- Host = atom()
- Arg = silent | verbose

As `world/0,1`, but the hosts are given as argument instead of being read from `.hosts.erlang`.

Files

The `.hosts.erlang` file consists of a number of host names written as Erlang terms. It is looked for in the current work directory, the user's home directory, and `$OTP_ROOT` (the root directory of Erlang/OTP), in that order.

The format of the `.hosts.erlang` file must be one host name per line. The host names must be within quotes as shown in the following example:

```
'super.eua.ericsson.se'.
'renat.eua.ericsson.se'.
'grouse.eua.ericsson.se'.
'gauffin1.eua.ericsson.se'.
^ (new line)
```

net_kernel

Erlang Module

The net kernel is a system process, registered as `net_kernel`, which must be running for distributed Erlang to work. The purpose of this process is to implement parts of the BIFs `spawn/4` and `spawn_link/4`, and to provide monitoring of the network.

An Erlang node is started using the command line flag `-name` or `-sname`:

```
$ erl -sname foobar
```

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

```
1> net_kernel:start([foobar, shortnames]).
{ok,<0.64.0>}
(foobar@gringotts)2>
```

If the node is started with the command line flag `-sname`, the node name will be `foobar@Host`, where `Host` is the short name of the host (not the fully qualified domain name). If started with the `-name` flag, `Host` is the fully qualified domain name. See `erl(1)`.

Normally, connections are established automatically when another node is referenced. This functionality can be disabled by setting the Kernel configuration parameter `dist_auto_connect` to `false`, see kernel(6) [page ??]. In this case, connections must be established explicitly by calling `net_kernel:connect_node/1`.

Which nodes are allowed to communicate with each other is handled by the magic cookie system, see [Distributed Erlang] in the Erlang Reference Manual.

Exports

`allow(Nodes) -> ok | error`

Types:

- Nodes = [node()]

Limits access to the specified set of nodes. Any access attempts made from (or to) nodes not in `Nodes` will be rejected.

Returns `error` if any element in `Nodes` is not an atom.

`connect_node(Node) -> true | false | ignored`

Types:

- Node = node()

Establishes a connection to Node. Returns true if successful, false if not, and ignored if the local node is not alive.

```
monitor_nodes(Flag) -> ok | Error
monitor_nodes(Flag, Options) -> ok | Error
```

Types:

- Flag = true | false
- Options = [Option]
- Option – see below
- Error = error | {error, term()}

The calling process subscribes or unsubcscribes to node status change messages. A nodeup message is delivered to all subscribing process when a new node is connected, and a nodedown message is delivered when a node is disconnected.

If Flag is true, a new subscription is started. If Flag is false, all previous subscriptions – started with the same Options – are stopped. Two option lists are considered the same if they contain the same set of options.

As of kernel version 2.11.4, and erts version 5.5.4, the following is guaranteed:

- nodeup messages will be delivered before delivery of any message from the remote node passed through the newly established connection.
- nodedown messages will not be delivered until all messages from the remote node that have been passed through the connection have been delivered.

Note, that this is *not* guaranteed for kernel versions before 2.11.4.

As of kernel version 2.11.4 subscribtions can also be made before the net_kernel server has been started, i.e., net_kernel:monitor_nodes/[1,2] does not return ignored.

The format of the node status change messages depends on Options. If Options is [], which is the default, the format is:

```
{nodeup, Node} | {nodedown, Node}
  Node = node()
```

If Options /= [], the format is:

```
{nodeup, Node, InfoList} | {nodedown, Node, InfoList}
  Node = node()
  InfoList = [{Tag, Val}]
```

InfoList is a list of tuples. Its contents depends on Options, see below.

Also, when OptionList == [] only visible nodes, that is, nodes that appear in the result of nodes/0 [page ??], are monitored.

Option can be any of the following:

{node_type, NodeType} Currently valid values for NodeType are:

visible Subscribe to node status change messages for visible nodes only. The tuple {node_type, visible} is included in InfoList.

hidden Subscribe to node status change messages for hidden nodes only. The tuple {node_type, hidden} is included in InfoList.

`all` Subscribe to node status change messages for both visible and hidden nodes. The tuple {`node_type, visible | hidden`} is included in `InfoList`.

`nodedown_reason` The tuple {`nodedown_reason, Reason`} is included in `InfoList` in `nodedown` messages. `Reason` can be:

`connection_setup_failed` The connection setup failed (after `nodeup` messages had been sent).

`no_network` No network available.

`net_kernel_terminated` The `net_kernel` process terminated.

`shutdown` Unspecified connection shutdown.

`connection_closed` The connection was closed.

`disconnect` The connection was disconnected (forced from the current node).

`net_tick_timeout` Net tick timeout.

`send_net_tick_failed` Failed to send net tick over the connection.

`get_status_failed` Status information retrieval from the `Port` holding the connection failed.

`get_net_ticktime() -> Res`

Types:

- Res = NetTicktime | {ongoing_change_to, NetTicktime}
- NetTicktime = int()

Gets `net_ticktime` (see kernel(6) [page ??]).

Currently defined return values (`Res`):

`NetTicktime` `net_ticktime` is `NetTicktime` seconds.

{`ongoing_change_to, NetTicktime`} `net_kernel` is currently changing `net_ticktime` to `NetTicktime` seconds.

`set_net_ticktime(NetTicktime) -> Res`

`set_net_ticktime(NetTicktime, TransitionPeriod) -> Res`

Types:

- NetTicktime = int() > 0
- TransitionPeriod = int() >= 0
- Res = unchanged | change_initiated | {ongoing_change_to, NewNetTicktime}
- NewNetTicktime = int() > 0

Sets `net_ticktime` (see kernel(6) [page ??]) to `NetTicktime` seconds.
`TransitionPeriod` defaults to 60.

Some definitions:

The minimum transition traffic interval (`MTTI`) `minimum(NetTicktime, PreviousNetTicktime)*1000 div 4` milliseconds.

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

If 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, net_kernel will ensure that there will be outgoing traffic on all connections at least every MTTI millisecond.

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

Returns one of the following:

`unchanged` net_ticktime already had the value of NetTicktime and was left unchanged.

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

{`ongoing_change_to, NewNetTicktime`} The request was *ignored*; because, net_kernel was busy changing net_ticktime to NewTicktime seconds.

```
start([Name]) -> {ok, pid()} | {error, Reason}
start([Name, NameType]) -> {ok, pid()} | {error, Reason}
start([Name, NameType, Ticktime]) -> {ok, pid()} | {error, Reason}
```

Types:

- Name = atom()
- NameType = shortnames | longnames
- Reason = {already_started, pid()} | term()

Note that the argument is a list with exactly one, two or three arguments. NameType defaults to longnames and Ticktime to 15000.

Turns a non-distributed node into a distributed node by starting net_kernel and other necessary processes.

```
stop() -> ok | {error, not_allowed | not_found}
```

Turns a distributed node into a non-distributed node. For other nodes in the network, this is the same as the node going down. Only possible when the net kernel was started using start/1, otherwise returns {error, not_allowed}. Returns {error, not_found} if the local node is not alive.

OS

Erlang Module

The functions in this module are operating system specific. Careless use of these functions will result in programs that will only run on a specific platform. On the other hand, with careful use these functions can be of help in enabling a program to run on most platforms.

Exports

`cmd(Command) -> string()`

Types:

- Command = string() | atom()

Executes `Command` in a command shell of the target OS, captures the standard output of the command and returns this result as a string. This function is a replacement of the previous `unix:cmd/1`; on a Unix platform they are equivalent.

Examples:

```
LsOut = os:cmd("ls"), % on unix platform
DirOut = os:cmd("dir"), % on Win32 platform
```

Note that in some cases, standard output of a command when called from another program (for example, `os:cmd/1`) may differ, compared to the standard output of the command when called directly from an OS command shell.

`find_executable(Name) -> Filename | false`

`find_executable(Name, Path) -> Filename | false`

Types:

- Name = 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 (that is, the environment variable PATH on Unix and Windows).

`Path`, if given, should conform to the syntax of execution paths on the operating system. The absolute filename of the executable program `Name` is returned, or `false` if the program was not found.

`getenv() -> [string()]`

Returns a list of all environement variables. Each environment variable is given as a single string on the format `"VarName=Value"`, where `VarName` is the name of the variable and `Value` its value.

`getenv(VarName) -> Value | false`

Types:

- VarName = string()
- Value = string()

Returns the `Value` of the environment variable `VarName`, or `false` if the environment variable is undefined.

`getpid() -> Value`

Types:

- Value = string()

Returns the process identifier of the current Erlang emulator in the format most commonly used by the operating system environment. `Value` is returned as a string containing the (usually) numerical identifier for a process. On Unix, this is typically the return value of the `getpid()` system call. On VxWorks, `Value` contains the task id (decimal notation) of the Erlang task. On Windows, the process id as returned by the `GetCurrentProcessId()` system call is used.

`putenv(VarName, Value) -> true`

Types:

- VarName = string()
- Value = string()

Sets a new `Value` for the environment variable `VarName`.

`type() -> {Osfamily, Osname} | Osfamily`

Types:

- Osfamily = win32 | unix | vxworks
- Osname = atom()

Returns the `Osfamily` and, in some cases, `Osname` of the current operating system.

On Unix, `Osname` will have same value as `uname -s` returns, but in lower case. For example, on Solaris 1 and 2, it will be `sunos`.

In Windows, `Osname` will be either `nt` (on Windows NT), or `windows` (on Windows 95).

On VxWorks the OS family alone is returned, that is `vxworks`.

> **Note:**
> Think twice before using this function. Use the `filename` module if you want to inspect or build file names in a portable way. Avoid matching on the `Osname` atom.

`version() -> {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) -> m2:g(X).
```

`m2:g(X)` becomes a call to `foo.bar.m2` If this is not what was intended, the call can be written explicitly, as in

```
-module(foo.bar.m1).
-export([f/1]).

f(X) -> fee.fie.foe.m2:g(X).
```

Because the called module is given with an explicit package name, no expansion is done in this case.

If a module from another package is used repeatedly in a module, an import declaration can make life easier:

```
-module(foo.bar.m1).
-export([f/1, g/1]).
-import(fee.fie.foe.m2).

f(X) -> m2:g(X).
g(X) -> m2:h(X).
```

will make the calls to `m2` refer to `fee.fie.foe.m2`. More generally, a declaration `-import(Package.Module).` will cause calls to `Module` to be expanded to `Package.Module`.

Old-style function imports work as normal (but full module names must be used); e.g.:

```
-import(fee.fie.foe.m2, [g/1, h/1]).
```

however, it is probably better to avoid this form of import altogether in new code, since it makes it hard to see what calls are really "remote".

If it is necessary to call a module in the top-level package from within a named package, the module name can be written either with an initial period as in e.g. "`.lists`", or with an empty initial atom, as in "`''.lists`". However, the best way is to use an import declaration - this is most obvious to the eye, and makes sure we don't forget adding a period somewhere:

```
-module(foo.bar.fred).
-export([f/1]).
-import(lists).

f(X) -> lists:reverse(X).
```

The dot-syntax for module names can be used in any expression. All segments must be constant atoms, and the result must be a well-formed package/module name. E.g.:

```
spawn(foo.bar.fred, f, [X])
```

is equivalent to `spawn('foo.bar.fred', f, [X])`.

The Erlang Shell

The shell also automatically expands remote calls, however currently no expansions are made by default. The user can change the behaviour by using the `import/1` shell function (or its abbreviation `use/1`). E.g.:

```
1> import(foo.bar.m).
ok
2> m:f().
```

will evaluate `foo.bar.m:f()`. If a new import is made of the same name, this overrides any previous import. (It is likely that in the future, some system packages will be pre-imported.)

In addition, the 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> 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 ??], 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

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

Types:

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

Evaluates `apply(Module, Function, Args)` on the node `Node` and returns the corresponding value `Res`, or `{badrpc, Reason}` if the call fails.

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

Types:

- Node = node()
- Module = Function = atom()
- Args = [term()]
- Res = term()
- Reason = timeout | term()
- Timeout = int() | infinity

Evaluates `apply(Module, Function, Args)` on the node `Node` and returns the corresponding value `Res`, or `{badrpc, Reason}` if the call fails. `Timeout` is a timeout value in milliseconds. If the call times out, `Reason` is `timeout`.

If the reply arrives after the call times out, no message will contaminate the caller's message queue, since this function spawns off a middleman process to act as (a void) destination for such an orphan reply. This feature also makes this function more expensive than `call/4` at the caller's end.

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

Types:

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

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

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

Types:

- Node = node()
- Module = Function = atom()
- Args = [term()]
- Timeout = int() | infinity
- Res = term()
- Reason = term()

Like `block_call/4`, but with a timeout value in the same manner as `call/5`.

`async_call(Node, Module, Function, Args) -> Key`

Types:

- Node = node()
- Module = Function = atom()
- Args = [term()]
- Key – see below

Implements *call streams with promises*, a type of RPC which does not suspend the caller until the result is finished. Instead, a key is returned which can be used at a later stage to collect the value. The key can be viewed as a promise to deliver the answer.

In this case, the key `Key` is returned, which can be used in a subsequent call to `yield/1` or `nb_yield/1,2` to retrieve the value of evaluating `apply(Module, Function, Args)` on the node `Node`.

`yield(Key) -> Res | {badrpc, Reason}`

Types:

- Key – see async_call/4
- Res = term()
- Reason = term()

Returns the promised answer from a previous `async_call/4`. If the answer is available, it is returned immediately. Otherwise, the calling process is suspended until the answer arrives from `Node`.

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

Types:

- Key – see async_call/4
- Val = Res | {badrpc, Reason}
- Res = term()
- Reason = term()

Equivalent to `nb_yield(Key, 0)`.

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

Types:

- Key – see async_call/4
- Timeout = int() | infinity
- Val = Res | {badrpc, Reason}
- Res = term()
- Reason = term()

This is a non-blocking version of `yield/1`. It returns the tuple `{value, Val}` when the computation has finished, or `timeout` when `Timeout` milliseconds has elapsed.

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

Types:

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

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

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

Types:

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

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

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

Types:

- Module = Function = atom()
- Args = [term()]
- Timeout = int() | infinity
- ResL = [term()]
- BadNodes = [node()]

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

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

Types:

- Nodes = [node()]
- Module = Function = atom()
- Args = [term()]
- Timeout = int() | infinity
- ResL = [term()]
- BadNodes = [node()]

In contrast to an RPC, a multicall is an RPC which is sent concurrently from one client to multiple servers. This is useful for collecting some information from a set of nodes, or for calling a function on a set of nodes to achieve some side effects. It is semantically the same as iteratively making a series of RPCs on all the nodes, but the multicall is faster as all the requests are sent at the same time and are collected one by one as they come back.

The function evaluates `apply(Module, Function, Args)` on the specified nodes and collects the answers. It returns `{ResL, Badnodes}`, where `Badnodes` is a list of the nodes that terminated or timed out during computation, and `ResL` is a list of the return values. `Timeout` is a time (integer) in milliseconds, or `infinity`.

The following example is useful when new object code is to be loaded on all nodes in the network, and also indicates some side effects RPCs may produce:

```
%% Find object code for module Mod
{Mod, Bin, File} = code:get_object_code(Mod),

%% and load it on all nodes including this one
{ResL, _} = rpc:multicall(code, load_binary, [Mod, Bin, File,]),

%% and then maybe check the ResL list.
```

`cast(Node, Module, Function, Args) -> void()`

Types:

- Node = node()
- Module = Function = atom()
- Args = [term()]

Evaluates `apply(Module, Function, Args)` on the node `Node`. No response is delivered and the calling process is not suspended until the evaluation is complete, as is the case with `call/4,5`.

`eval_everywhere(Module, Funtion, Args) -> void()`

Types:

- Module = Function = atom()
- Args = [term()]

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

`eval_everywhere(Nodes, Module, Function, Args) -> void()`

Types:

- Nodes = [node()]
- Module = Function = atom()

- Args = [term()]

Evaluates `apply(Module, Function, Args)` on the specified nodes. No answers are collected.

`abcast(Name, Msg) -> void()`

Types:

- Name = atom()
- Msg = term()

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

`abcast(Nodes, Name, Msg) -> void()`

Types:

- Nodes = [node()]
- Name = atom()
- Msg = term()

Broadcasts the message `Msg` asynchronously to the registered process `Name` on the specified nodes.

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

Types:

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

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

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

Types:

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

Broadcasts the message `Msg` synchronously to the registered process `Name` on the specified nodes.

Returns `{GoodNodes, BadNodes}`, where `GoodNodes` is the list of nodes which have `Name` as a registered process.

The function is synchronous in the sense that it is known that all servers have received the message when the call returns. It is not possible to know that the servers have actually processed the message.

Any further messages sent to the servers, after this function has returned, will be received by all servers after this message.

`server_call(Node, Name, ReplyWrapper, Msg) -> Reply | {error, Reason}`

Types:

- Node = node()
- Name = atom()

- ReplyWrapper = Msg = Reply = term()
- Reason = term()

This function can be used when interacting with a server called `Name` at node `Node`. It is assumed that the server receives messages in the format `{From, Msg}` and replies using `From ! {ReplyWrapper, Node, Reply}`. This function makes such a server call and ensures that the entire call is packed into an atomic transaction which either succeeds or fails. It never hangs, unless the server itself hangs.

The function returns the answer `Reply` as produced by the server `Name`, or `{error, Reason}`.

`multi_server_call(Name, Msg) -> {Replies, BadNodes}`

Types:

- Name = atom()
- Msg = term()
- Replies = [Reply]
- Reply = term()
- BadNodes = [node()]

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

`multi_server_call(Nodes, Name, Msg) -> {Replies, BadNodes}`

Types:

- Nodes = [node()]
- Name = atom()
- Msg = term()
- Replies = [Reply]
- Reply = term()
- BadNodes = [node()]

This function can be used when interacting with servers called `Name` on the specified nodes. It is assumed that the servers receive messages in the format `{From, Msg}` and reply using `From ! {Name, Node, Reply}`, where `Node` is the name of the node where the server is located. The function returns `{Replies, Badnodes}`, where `Replies` is a list of all `Reply` values and `BadNodes` is a list of the nodes which did not exist, or where the server did not exist, or where the server terminated before sending any reply.

`safe_multi_server_call(Name, Msg) -> {Replies, BadNodes}`
`safe_multi_server_call(Nodes, Name, Msg) -> {Replies, BadNodes}`

> **Warning:**
> This function is deprecated. Use `multi_server_call/2,3` instead.

In Erlang/OTP R6B and earlier releases, `multi_server_call/2,3` could not handle the case where the remote node exists, but there is no server called `Name`. Instead this function had to be used. In Erlang/OTP R7B and later releases, however, the functions are equivalent, except for this function being slightly slower.

`parallel_eval(FuncCalls) -> ResL`

Types:

- FuncCalls = [{Module, Function, Args}]
- Module = Function = atom()
- Args = [term()]
- ResL = [term()]

For every tuple in `FuncCalls`, evaluates `apply(Module, Function, Args)` on some node in the network. Returns the list of return values, in the same order as in `FuncCalls`.

`pmap({Module, Function}, ExtraArgs, List2) -> List1`

Types:

- Module = Function = atom()
- ExtraArgs = [term()]
- List1 = [Elem]
- Elem = term()
- List2 = [term()]

Evaluates `apply(Module, Function, [Elem|ExtraArgs])`, for every element `Elem` in `List1`, in parallell. Returns the list of return values, in the same order as in `List1`.

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

Types:

- Pid = pid()
- Item, Info – see erlang:process_info/1

Location transparent version of the BIF `process_info/1`.

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

Types:

- Pid = pid()
- Item, Info – see erlang:process_info/1

Location transparent version of the BIF `process_info/2`.

seq_trace

Erlang Module

Sequential tracing makes it possible to trace all messages resulting from one initial message. Sequential tracing is completely independent of the ordinary tracing in Erlang, which is controlled by the `erlang:trace/3` BIF. See the chapter What is Sequential Tracing [page ??] 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(Token) -> PreviousToken`

Types:

- Token = PreviousToken = term() | []

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

```
OldToken = seq_trace:set_token([]), % set to empty and save
                                    % old value
% do something that should not be part of the trace
io:format("Exclude the signalling caused by this~n"),
seq_trace:set_token(OldToken), % activate the trace token again
...
```

Returns the previous value of the trace token.

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

Types:

- Component = label | serial | Flag
- Flag = send | 'receive' | print | timestamp

- Val = OldVal – see below

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

`set_token(label, Int)` The `label` component is an integer which identifies all events belonging to the same sequential trace. If several sequential traces can be active simultaneously, `label` is used to identify the separate traces. Default is 0.

`set_token(serial, SerialValue) SerialValue = {Previous, Current}`. The `serial` component contains counters which enables the traced messages to be sorted, should never be set explicitly by the user as these counters are updated automatically. Default is `{0, 0}`.

`set_token(send, Bool)` A trace token flag (`true | false`) which enables/disables tracing on message sending. Default is `false`.

`set_token('receive', Bool)` A trace token flag (`true | false`) which enables/disables tracing on message reception. Default is `false`.

`set_token(print, Bool)` A trace token flag (`true | false`) which enables/disables tracing on explicit calls to `seq_trace:print/1`. Default is `false`.

`set_token(timestamp, Bool)` A trace token flag (`true | false`) which enables/disables a timestamp to be generated for each traced event. Default is `false`.

`get_token() -> TraceToken`

Types:

- TraceToken = term() | []

Returns the value of the trace token for the calling process. If `[]` is returned, it means that tracing is not active. Any other value returned is the value of an active trace token. The value returned can be used as input to the `set_token/1` function.

`get_token(Component) -> {Component, Val}`

Types:

- Component = label | serial | Flag
- Flag = send | 'receive' | print | timestamp
- Val – see set_token/2

Returns the value of the trace token component `Component`. See set_token/2 [page ??] for possible values of `Component` and `Val`.

`print(TraceInfo) -> void()`

Types:

- TraceInfo = term()

Puts the Erlang term `TraceInfo` into the sequential trace output if the calling process currently is executing within a sequential trace and the `print` flag of the trace token is set.

`print(Label, TraceInfo) -> void()`

Types:

- Label = int()
- TraceInfo = term()

Same as `print/1` with the additional condition that `TraceInfo` is output only if `Label` is equal to the label component of the trace token.

`reset_trace() -> void()`

Sets the trace token to empty for all processes on the local node. The process internal counters used to create the serial of the trace token is set to 0. The trace token is set to empty for all messages in message queues. Together this will effectively stop all ongoing sequential tracing in the local node.

`set_system_tracer(Tracer) -> OldTracer`

Types:

- Tracer = OldTracer = pid() | port() | false

Sets the system tracer. The system tracer can be either a process or port denoted by `Tracer`. Returns the previous value (which can be `false` if no system tracer is active).

Failure: {`badarg, Info`}} if `Pid` is not an existing local pid.

`get_system_tracer() -> Tracer`

Types:

- Tracer = pid() | port() | false

Returns the pid or port identifier of the current system tracer or `false` if no system tracer is activated.

Trace Messages Sent To the System Tracer

The format of the messages are:

```
{seq_trace, Label, SeqTraceInfo, TimeStamp}
```

or

```
{seq_trace, Label, SeqTraceInfo}
```

depending on whether the `timestamp` flag of the trace token is set to `true` or `false`. Where:

```
Label = int()
TimeStamp = {Seconds, Milliseconds, Microseconds}
  Seconds = Milliseconds = Microseconds = int()
```

The `SeqTraceInfo` can have the following formats:

{`send, Serial, From, To, Message`} Used when a process `From` with its trace token flag `print` set to `true` has sent a message.

{`'receive', Serial, From, To, Message`} Used when a process `To` receives a message with a trace token that has the `'receive'` flag set to `true`.

{`print, Serial, From, _, Info`} Used when a process `From` has called `seq_trace:print(Label, TraceInfo)` and has a trace token with the `print` flag set to `true` and `label` set to `Label`.

`Serial` is a tuple {`PreviousSerial, ThisSerial`}, where the first integer `PreviousSerial` denotes the serial counter passed in the last received message which carried a trace token. If the process is the first one in a new sequential trace, `PreviousSerial` is set to the value of the process internal "trace clock". The second integer `ThisSerial` is the serial counter that a process sets on outgoing messages and it is based on the process internal "trace clock" which is incremented by one before it is attached to the trace token in the message.

What is Sequential Tracing

Sequential tracing is a way to trace a sequence of messages sent between different local or remote processes, where the sequence is initiated by one single message. In short it works like this:

Each process has a *trace token*, which can be empty or not empty. When not empty the trace token can be seen as the tuple {`Label, Flags, Serial, From`}. The trace token is passed invisibly with each message.

In order to start a sequential trace the user must explicitly set the trace token in the process that will send the first message in a sequence.

The trace token of a process is set each time the process matches a message in a receive statement, according to the trace token carried by the received message, empty or not.

On each Erlang node a process can be set as the *system tracer*. This process will receive trace messages each time a message with a trace token is sent or received (if the trace token flag `send` or `'receive'` is set). The system tracer can then print each trace event, write it to a file or whatever suitable.

> Note:
> The system tracer will only receive those trace events that occur locally within the Erlang node. To get the whole picture of a sequential trace that involves processes on several Erlang nodes, the output from the system tracer on each involved node must be merged (off line).

In the following sections Sequential Tracing and its most fundamental concepts are described.

Trace Token

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

The current token of a process is set in two ways, either

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 is called `serial`. It consists of two integers `Previous` and `Current`. The purpose is to uniquely identify each traced event within a trace sequence and to order the messages chronologically and in the different branches if any.

The algorithm for updating `Serial` can be described as follows:

Let each process have two counters `prev_cnt` and `curr_cnt` which both are set to 0 when a process is created. The counters are updated at the following occasions:

- *When the process is about to send a message and the trace token is not empty.*
  Let the serial of the trace token be `tprev` and `tcurr`.
  ```
  curr_cnt := curr_cnt + 1
  tprev := prev_cnt
  tcurr := curr_cnt
  ```
  The trace token with `tprev` and `tcurr` is then passed along with the message.
- *When the process calls* `seq_trace:print(Label, Info)`, *Label matches the label part of the trace token and the trace token print flag is true.*
  The same algorithm as for send above.
- *When a message is received and contains a nonempty trace token.*
  The process trace token is set to the trace token from the message.
  Let the serial of the trace token be `tprev` and `tcurr`.
  ```
  if (curr_cnt < tcurr )
    curr_cnt := tcurr
  prev_cnt := 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.

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 ??]

zlib

Erlang Module

The zlib module provides an API for the zlib library (http://www.zlib.org). It is used to compress and decompress data. The data format is described by RFCs 1950 to 1952.

A typical (compress) usage looks like:

```
Z = zlib:open(),
ok = zlib:deflateInit(Z,default),

Compress = fun(end_of_data, _Cont) -> [];
              (Data, Cont) ->
                 [zlib:deflate(Z, Data)|Cont(Read(),Cont)]
           end,
Compressed = Compress(Read(),Compress),
Last = zlib:deflate(Z, [], finish),
ok = zlib:deflateEnd(Z),
zlib:close(Z),
list_to_binary([Compressed|Last])
```

In all functions errors, {'EXIT',{Reason,Backtrace}}, might be thrown, where Reason describes the error. Typical reasons are:

`badarg` Bad argument

`data_error` The data contains errors

`stream_error` Inconsistent stream state

`einval` Bad value or wrong function called

{`need_dictionary`,`Adler32`} See `inflate/2`

DATA TYPES

```
iodata = iolist() | binary()

iolist = [char() | binary() | iolist()]
  a binary is allowed as the tail of the list

zstream = a zlib stream, see open/0
```

Exports

`open() -> Z`

Types:

- Z = zstream()

Open a zlib stream.

`close(Z) -> ok`

Types:

- Z = zstream()

Closes the stream referenced by `Z`.

`deflateInit(Z) -> ok`

Types:

- Z = zstream()

Same as `zlib:deflateInit(Z, default)`.

`deflateInit(Z, Level) -> ok`

Types:

- Z = zstream()
- Level = none | default | best_speed | best_compression | 0..9

Initialize a zlib stream for compression.

`Level` decides the compression level to be used, 0 (`none`), gives no compression at all, 1 (`best_speed`) gives best speed and 9 (`best_compression`) gives best compression.

`deflateInit(Z, Level, Method, WindowBits, MemLevel, Strategy) -> ok`

Types:

- Z = zstream()
- Level = none | default | best_speed | best_compression | 0..9
- Method = deflated
- WindowBits = 9..15|-9..-15
- MemLevel = 1..9
- Strategy = default|filtered|huffman_only

Initiates a zlib stream for compression.

The `Level` parameter decides the compression level to be used, 0 (`none`), gives no compression at all, 1 (`best_speed`) gives best speed and 9 (`best_compression`) gives best compression.

The `Method` parameter decides which compression method to use, currently the only supported method is `deflated`.

The `WindowBits` parameter is the base two logarithm of the window size (the size of the history buffer). It should be in the range 9 through 15. Larger values of this parameter result in better compression at the expense of memory usage. The default value is 15 if `deflateInit/2`. A negative `WindowBits` value supresses the zlib header (and checksum) from the stream. Note that the zlib source mentions this only as a undocumented feature.

The `MemLevel` parameter specifies how much memory should be allocated for the internal compression state. `MemLevel=1` uses minimum memory but is slow and reduces compression ratio; `MemLevel=9` uses maximum memory for optimal speed. The default value is 8.

The `Strategy` parameter is used to tune the compression algorithm. Use the value `default` for normal data, `filtered` for data produced by a filter (or predictor), or `huffman_only` to force Huffman encoding only (no string match). Filtered data consists mostly of small values with a somewhat random distribution. In this case, the compression algorithm is tuned to compress them better. The effect of `filtered`is to force more Huffman coding and less string matching; it is somewhat intermediate between `default` and `huffman_only`. The `Strategy` parameter only affects the compression ratio but not the correctness of the compressed output even if it is not set appropriately.

`deflate(Z, Data) -> Compressed`

Types:

- Z = zstream()
- Data = iodata()
- Compressed = iolist()

Same as `deflate(Z, Data, none)`.

`deflate(Z, Data, Flush) ->`

Types:

- Z = zstream()
- Data = iodata()
- Flush = none | sync | full | finish
- Compressed = iolist()

`deflate/3` compresses as much data as possible, and stops when the input buffer becomes empty. It may introduce some output latency (reading input without producing any output) except when forced to flush.

If the parameter `Flush` is set to `sync`, all pending output is flushed to the output buffer and the output is aligned on a byte boundary, so that the decompressor can get all input data available so far. Flushing may degrade compression for some compression algorithms and so it should be used only when necessary.

If Flush is set to full, all output is flushed as with sync, and the compression state is reset so that decompression can restart from this point if previous compressed data has been damaged or if random access is desired. Using full too often can seriously degrade the compression.

If the parameter Flush is set to finish, pending input is processed, pending output is flushed and deflate/3 returns. Afterwards the only possible operations on the stream are deflateReset/1 or deflateEnd/1.

Flush can be set to finish immediately after deflateInit if all compression is to be done in one step.

```
zlib:deflateInit(Z),
B1 = zlib:deflate(Z,Data),
B2 = zlib:deflate(Z,<< >>,finish),
zlib:deflateEnd(Z),
list_to_binary([B1,B2]),
```

`deflateSetDictionary(Z, Dictionary) -> Adler32`

Types:

- Z = zstream()
- Dictionary = binary()
- Adler32 = integer()

Initializes the compression dictionary from the given byte sequence without producing any compressed output. This function must be called immediately after deflateInit/[1|2|6] or deflateReset/1, before any call of deflate/3. The compressor and decompressor must use exactly the same dictionary (see inflateSetDictionary/2). The adler checksum of the dictionary is returned.

`deflateReset(Z) -> ok`

Types:

- Z = zstream()

This function is equivalent to deflateEnd/1 followed by deflateInit/[1|2|6], but does not free and reallocate all the internal compression state. The stream will keep the same compression level and any other attributes.

`deflateParams(Z, Level, Strategy) -> ok`

Types:

- Z = zstream()
- Level = none | default | best_speed | best_compression | 0..9
- Strategy = default|filtered|huffman_only

Dynamically update the compression level and compression strategy. The interpretation of `Level` and `Strategy` is as in `deflateInit/6`. This can be used to switch between compression and straight copy of the input data, or to switch to a different kind of input data requiring a different strategy. If the compression level is changed, the input available so far is compressed with the old level (and may be flushed); the new level will take effect only at the next call of `deflate/3`.

Before the call of deflateParams, the stream state must be set as for a call of `deflate/3`, since the currently available input may have to be compressed and flushed.

`deflateEnd(Z) -> ok`

Types:

- Z = zstream()

End the deflate session and cleans all data used. Note that this function will throw an `data_error` exception if the last call to `deflate/3` was not called with `Flush` set to `finish`.

`inflateInit(Z) -> ok`

Types:

- Z = zstream()

Initialize a zlib stream for decompression.

`inflateInit(Z, WindowBits) -> ok`

Types:

- Z = zstream()
- WindowBits = 9..15|-9..-15

Initialize decompression session on zlib stream.

The `WindowBits` parameter is the base two logarithm of the maximum window size (the size of the history buffer). It should be in the range 9 through 15. The default value is 15 if `inflateInit/1` is used. If a compressed stream with a larger window size is given as input, inflate() will throw the `data_error` exception. A negative `WindowBits` value makes zlib ignore the zlib header (and checksum) from the stream. Note that the zlib source mentions this only as a undocumented feature.

`inflate(Z, Data) -> DeCompressed`

Types:

- Z = zstream()
- Data = iodata()
- DeCompressed = iolist()

`inflate/2` decompresses as much data as possible. It may some introduce some output latency (reading input without producing any output).

If a preset dictionary is needed at this point (see `inflateSetDictionary` below), `inflate/2` throws a {`need_dictionary`,`Adler`} exception where `Adler` is the adler32 checksum of the dictionary chosen by the compressor.

`inflateSetDictionary(Z, Dictionary) -> ok`

Types:

- Z = zstream()
- Dictionary = binary()

Initializes the decompression dictionary from the given uncompressed byte sequence. This function must be called immediately after a call of `inflate/2` if this call threw a {`need_dictionary`,`Adler`} exception. The dictionary chosen by the compressor can be determined from the Adler value thrown by the call to `inflate/2`. The compressor and decompressor must use exactly the same dictionary (see `deflateSetDictionary/2`).

Example:

```
unpack(Z, Compressed, Dict) ->
     case catch zlib:inflate(Z, Compressed) of
          {'EXIT',{{need_dictionary,DictID},_}} ->
                 zlib:inflateSetDictionary(Z, Dict),
                 Uncompressed = zlib:inflate(Z, []);
          Uncompressed ->
                 Uncompressed
     end.
```

`inflateReset(Z) -> ok`

Types:

- Z = zstream()

This function is equivalent to `inflateEnd/1` followed by `inflateInit/1`, but does not free and reallocate all the internal decompression state. The stream will keep attributes that may have been set by `inflateInit/[1|2]`.

`inflateEnd(Z) -> ok`

Types:

- Z = zstream()

End the inflate session and cleans all data used. Note that this function will throw a `data_error` exception if no end of stream was found (meaning that not all data has been uncompressed).

`setBufSize(Z, Size) -> ok`

Types:

- Z = zstream()
- Size = integer()

Sets the intermediate buffer size.

`getBufSize(Z) -> Size`

Types:

- Z = zstream()
- Size = integer()

Get the size of intermediate buffer.

`crc32(Z) -> CRC`

Types:

- Z = zstream()
- CRC = integer()

Get the current calulated CRC checksum.

`crc32(Z, Binary) -> CRC`

Types:

- Z = zstream()
- Binary = binary()
- CRC = integer()

Calculate the CRC checksum for `Binary`.

`crc32(Z, PrevCRC, Binary) -> CRC`

Types:

- Z = zstream()
- PrevCRC = integer()
- Binary = binary()
- CRC = integer()

Update a running CRC checksum for `Binary`. If `Binary` is the empty binary, this function returns the required initial value for the crc.

```
Crc = lists:foldl(fun(Bin,Crc0) ->
                      zlib:crc32(Z, Crc0, Bin),
                  end, zlib:crc32(Z,<< >>), Bins),
```

`adler32(Z, Binary) -> Checksum`

Types:

- Z = zstream()
- Binary = binary()
- Checksum = integer()

Calculate the Adler-32 checksum for `Binary`.

`adler32(Z, PrevAdler, Binary) -> Checksum`

Types:

- Z = zstream()
- PrevAdler = integer()
- Binary = binary()
- Checksum = integer()

Update a running Adler-32 checksum for `Binary`. If `Binary` is the empty binary, this function returns the required initial value for the checksum.

```
            Crc = lists:foldl(fun(Bin,Crc0) ->
                                  zlib:adler32(Z, Crc0, Bin),
                              end, zlib:adler32(Z,<< >>), Bins),
```

`compress(Binary) -> Compressed`

Types:

- Binary = Compressed = binary()

Compress a binary (with zlib headers and checksum).

`uncompress(Binary) -> Decompressed`

Types:

- Binary = Decompressed = binary()

Uncompress a binary (with zlib headers and checksum).

`zip(Binary) -> Compressed`

Types:

- Binary = Compressed = binary()

Compress a binary (without zlib headers and checksum).

`unzip(Binary) -> Decompressed`

Types:

- Binary = Decompressed = binary()

Uncompress a binary (without zlib headers and checksum).

`gzip(Data) -> Compressed`

Types:

- Binary = Compressed = binary()

Compress a binary (with gz headers and checksum).

`gunzip(Bin) -> Decompressed`

Types:

- Binary = Decompressed = binary()

Uncompress a binary (with gz headers and checksum).

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 called `Application.app` for each application `Application` in the system.

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

FILE SYNTAX

The application resource file should be called `Application.app` where `Application` is the name of the application. The file should be located in the `ebin` directory for the application.

It must contain one single Erlang term, which is called an *application specification*:

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

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

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

`Application` is the name of the application.

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

The functions in `systools` require more information. If they are used, the following keys are mandatory: `description`, `vsn`, `modules`, `registered` and `applications`. The other keys are ignored by `systools`.

`description` A one-line description of the application.

`id` Product identification, or similar.

`vsn` The version of the application.

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

`maxP` *Deprecated - will be ignored*
The maximum number of processes allowed in the application.

`maxT` The maximum time in milliseconds that the application is allowed to run. After the specified time the application will automatically terminate.

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

`included_applications` All applications which are included by this application. When this application is started, all included application will automatically be loaded, but not started, by the application controller. It is assumed that the topmost supervisor of the included application is started by a supervisor of this application.

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

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

`mod` Specifies the application callback module and a start argument, see `application(3)`.

The `mod` key is necessary for an application implemented as a supervision tree, or the application controller will not know how to start it. The `mod` key can be omitted for applications without processes, typically code libraries such as the application STDLIB.

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

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

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

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

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

SEE ALSO

application(3) [page ??], 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*