

Kernel Application (KERNEL)

version 2.7

Typeset in \LaTeX from SGML source using the DOCBUILDER 3.2.2 Document System.

Contents

1	Kernel Reference Manual	1
1.1	kernel	27
1.2	application	31
1.3	auth	40
1.4	code	43
1.5	disk_log	50
1.6	erl_boot_server	64
1.7	erl_ddll	66
1.8	erl_prim_loader	69
1.9	erlang	72
1.10	error_handler	109
1.11	error_logger	111
1.12	file	116
1.13	gen_tcp	133
1.14	gen_udp	137
1.15	global	139
1.16	global_group	144
1.17	heart	148
1.18	inet	150
1.19	init	157
1.20	net_adm	162
1.21	net_kernel	164
1.22	os	166
1.23	pg2	169
1.24	rpc	171
1.25	seq_trace	175
1.26	user	183
1.27	wrap_log_reader	184
1.28	app	186
1.29	config	189

Kernel Reference Manual

Short Summaries

- Application **kernel** [page 27] – The Kernel Application
- Erlang Module **application** [page 31] – Generic OTP application functions.
- Erlang Module **auth** [page 40] – The Erlang Network Authentication Server
- Erlang Module **code** [page 43] – Erlang Code Server
- Erlang Module **disk_log** [page 50] – A disk based term logging facility
- Erlang Module **erl_boot_server** [page 64] – Boot Server for Other Erlang Machines
- Erlang Module **erl_ddll** [page 66] – Dynamic Driver Loader and Linker
- Erlang Module **erl_prim_loader** [page 69] – The Low Level Erlang Loader.
- Erlang Module **erlang** [page 72] – The Erlang BIFs
- Erlang Module **error_handler** [page 109] – Default System Error Handler
- Erlang Module **error_logger** [page 111] – The Erlang Error Logger
- Erlang Module **file** [page 116] – File Interface Module
- Erlang Module **gen_tcp** [page 133] – Interface to TCP/IP sockets
- Erlang Module **gen_udp** [page 137] – Interface to UDP.
- Erlang Module **global** [page 139] – A Global Name Registration Facility
- Erlang Module **global_group** [page 144] – Grouping Nodes to Global Name Registration Groups
- Erlang Module **heart** [page 148] – Heartbeat Monitoring of an Erlang Runtime System.
- Erlang Module **inet** [page 150] – Access to TCP/IP protocols.
- Erlang Module **init** [page 157] – Called at System Start
- Erlang Module **net_adm** [page 162] – Various Erlang Net Administration Routines
- Erlang Module **net_kernel** [page 164] – Erlang Networking Kernel
- Erlang Module **os** [page 166] – Operating System Specific Functions
- Erlang Module **pg2** [page 169] – Distributed Named Process Groups
- Erlang Module **rpc** [page 171] – Remote Procedure Call Services
- Erlang Module **seq_trace** [page 175] – Sequential Tracing of Messages.
- Erlang Module **user** [page 183] – Standard I/O Server
- Erlang Module **wrap_log_reader** [page 184] – A function to read internally formatted wrap disk logs
- File **app** [page 186] – Application resource file.
- File **config** [page 189] – Configuration file.

kernel

No functions are exported.

application

The following functions are exported:

- `get_all_env()` -> `Env`
[page 32] Get the configuration parameters for an application.
- `get_all_env(Application)` -> `Env`
[page 32] Get the configuration parameters for an application.
- `get_all_key()` -> `{ok, Keys} | []`
[page 32] Get the application specification keys.
- `get_all_key(Application)` -> `{ok, Keys} | undefined`
[page 32] Get the application specification keys.
- `get_application()` -> `{ok, Application} | undefined`
[page 32] Get the name of an application containing a certain process or module.
- `get_application(Pid | Module)` -> `{ok, Application} | undefined`
[page 32] Get the name of an application containing a certain process or module.
- `get_env(Par)` -> `{ok, Val} | undefined`
[page 32] Get the value of a configuration parameter.
- `get_env(Application, Par)` -> `{ok, Val} | undefined`
[page 32] Get the value of a configuration parameter.
- `get_key(Key)` -> `{ok, Val} | undefined`
[page 33] Get the value of an application specification key.
- `get_key(Application, Key)` -> `{ok, Val} | undefined`
[page 33] Get the value of an application specification key.
- `load(ApplDescr)` -> `ok | {error, Reason}`
[page 33] Load an application.
- `load(ApplDescr, Distributed)` -> `ok | {error, Reason}`
[page 33] Load an application.
- `loaded_applications()` -> `[[Application, Description, Vsn]]`
[page 34] Get the currently loaded applications.
- `permit(Application, Bool)` -> `ok | {error, Reason}`
[page 34] Change an application's permission to run on a node.
- `set_env(Application, Par, Val)` -> `ok`
[page 35] Set the value of a configuration parameter.
- `start(Application)` -> `ok | {error, Reason}`
[page 35] Load and start an application.
- `start(Application, Type)` -> `ok | {error, Reason}`
[page 35] Load and start an application.
- `start_type()` -> `StartType | local`
[page 35] Get the start type of an ongoing application startup.
- `stop(Application)` -> `ok | {error, Reason}`
[page 36] Stop an application.

- `takeover(Application, Type) -> ok | {error, Reason}`
[page 36] Take over a distributed application.
- `unload(Application) -> ok | {error, Reason}`
[page 36] Unload an application.
- `which_applications() -> [{Application, Description, Vsn}]`
[page 37] Get the currently running applications.
- `Module:config_change(Changed, New, Removed) -> ok`
[page 37] Update the configuration parameters for an application.
- `Module:start(StartType, StartArgs) -> {ok, Pid} | {ok, Pid, State} | {error, Reason}`
[page 37] Start an application.
- `Module:start_phase(Phase, StartType, PhaseArgs) -> ok | {error, Reason}`
[page 38] Extended start of an application.
- `Module:prep_stop(State) -> NewState`
[page 38] Prepare an application for termination.
- `Module:stop(State)`
[page 39] Clean up before termination of an application.

auth

The following functions are exported:

- `start()`
[page 41] Start the auth server
- `stop()`
[page 41] Stop the auth server
- `open(Name)`
[page 41] Open up a server
- `is_auth(Node)`
[page 41] Return status of communication authorization
- `exists(Node)`
[page 41] Check if a node exists
- `cookie()`
[page 41] Read and set cookies
- `node_cookie(Node, Cookie)`
[page 41] Check if a cookie is known
- `node_cookie([Node, Cookie])`
[page 41] Check if a cookie is known
- `cookie([Cookie])`
[page 42] Set the distribution cookie for the local node

code

The following functions are exported:

- `start()` -> {ok, Pid} | {error, What}
[page 43] Start the code server.
- `start(Flags)` -> {ok, Pid} | {error, What}
[page 43] Start the code server.
- `start_link()` -> {ok, Pid} | {error, What}
[page 43] Start and links to the code server.
- `start_link(Flags)` -> {ok, Pid} | {error, What}
[page 43] Start and links to the code server.
- `set_path(DirList)` -> true | {error, What}
[page 44] Set the code server search path.
- `get_path()` -> Path
[page 44] Return the current path of the code server.
- `add_path(Dir)` -> true | {error, What}
[page 44] Add a directory to the end of path.
- `add_pathz(Dir)` -> true | {error, What}
[page 44] Add a directory to the end of path.
- `add_patha(Dir)` -> true | {error, What}
[page 44] Add a directory to the beginning of path.
- `add_paths(DirList)` -> ok
[page 44] Add directories to the end of path.
- `add_pathsz(DirList)` -> ok
[page 44] Add directories to the end of path.
- `add_pathsa(DirList)` -> ok
[page 45] Add directories to the beginning of path.
- `del_path(NameDir)` -> true | false | {error, What}
[page 45] Delete a directory from the path.
- `replace_path(Name, Dir)` -> true | {error, What}
[page 45] Replace a directory with another in the path.
- `load_file(Module)` -> {module, Module} | {error, What}
[page 45] Load a module (residing in File).
- `load_abs(File)` -> {module, Module} | {error, What}
[page 45] Load a module (residing in File).
- `ensure_loaded(Module)` -> {module, Module} | {error, What}
[page 46] Try to ensure that a module is loaded.
- `delete(Module)` -> true | false
[page 46] Delete the code in Module.
- `purge(Module)` -> true | false
[page 46] Purges the code in Module.
- `soft_purge(Module)` -> true | false
[page 46] Purge the code in Module if no process uses it.
- `is_loaded(Module)` -> {file, Loaded} | false
[page 46] Test if Module is loaded.

- `all_loaded()` -> `[LoadMod]`
[page 47] Get all loaded modules.
- `load_binary(Module, File, Binary)` -> `{module, Module} | {error, What}`
[page 47] Load object code as a binary.
- `stop()` -> `stopped`
[page 47] Stop the code server.
- `root_dir()` -> `RootDir`
[page 47] Return the root directory of Erlang/OTP.
- `lib_dir()` -> `LibDir`
[page 47] Return the library directory.
- `lib_dir(Name)` -> `LibDir | {error, What}`
[page 47] Return the directory for name.
- `compiler_dir()` -> `CompDir`
[page 48] Return the compiler directory.
- `priv_dir(Name)` -> `PrivDir | {error, What}`
[page 48] Return the priv directory for name.
- `get_object_code(Module)` -> `{Module, Bin, AbsFileName} | error`
[page 48] Get the object code for a module.
- `objfile_extension()` -> `Ext`
[page 48] Return the object code file extension.
- `stick_dir(Dir)` -> `ok | {error, term()}`
[page 48] Mark a directory as 'sticky'.
- `unstick_dir(Dir)` -> `ok | {error, term()}`
[page 49] Mark a directory as 'non-sticky'.
- `which(Module)` -> `WhichFile`
[page 49] Return the directory of a module.
- `clash()` -> `ok`
[page 49] Search for modules with identical names.

disk_log

The following functions are exported:

- `accessible_logs()` -> `{[LocalLog], [DistributedLog]}`
[page 52] Return the accessible disk logs on the current node.
- `alog(Log, Term)` -> `ok | {error, Reason}`
[page 52] Asynchronously log an item onto a disk log.
- `balog(Log, Bytes)` -> `ok | {error, Reason}`
[page 52] Asynchronously log an item onto a disk log.
- `alog_terms(Log, TermList)` -> `ok | {error, Reason}`
[page 52] Asynchronously log several items onto a disk log.
- `balog_terms(Log, BytesList)` -> `ok | {error, Reason}`
[page 52] Asynchronously log several items onto a disk log.
- `block(Log)`
[page 53] Block a disk log.

- `block(Log, QueueLogRecords) -> ok | {error, Reason}`
[page 53] Block a disk log.
- `change_header(Log, Header) -> ok | {error, Reason}`
[page 53] Change the head or head_func option for an owner of a disk log.
- `change_notify(Log, Owner, Notify) -> ok | {error, Reason}`
[page 53] Change the notify option for an owner of a disk log.
- `change_size(Log, Size) -> ok | {error, Reason}`
[page 53] Change the size of an open disk log.
- `chunk(Log, Continuation)`
[page 54] Read a chunk of objects written to a disk log.
- `chunk(Log, Continuation, N) -> {Continuation2, Terms} | {Continuation2, Terms, Badbytes} | eof | {error, Reason}`
[page 54] Read a chunk of objects written to a disk log.
- `chunk_info(Continuation) -> InfoList | {error, Reason}`
[page 55] Return information about a chunk continuation of a disk log.
- `chunk_step(Log, Continuation, Step) -> {ok, Continuation2} | {error, Reason}`
[page 55] Step forward or backward among the wrap log files of a disk log.
- `close(Log) -> ok | {error, Reason}`
[page 55] Close a disk log.
- `format_error(Error) -> character_list()`
[page 56] Return an English description of a disk log error reply.
- `inc_wrap_file(Log) -> ok | {error, Reason}`
[page 56] Change to the next wrap log file of a disk log.
- `info(Log) -> InfoList | {error, no_such_log}`
[page 56] Return information about a disk log.
- `lclose(Log) -> ok | {error, Reason}`
[page 57] Close a disk log on one node.
- `lclose(Log, Node) -> ok | {error, Reason}`
[page 57] Close a disk log on one node.
- `log(Log, Term) -> ok | {error, Reason}`
[page 58] Log an item onto a disk log.
- `blog(Log, Bytes) -> ok | {error, Reason}`
[page 58] Log an item onto a disk log.
- `log_terms(Log, TermList) -> ok | {error, Reason}`
[page 58] Log several items onto a disk log.
- `blog_terms(Log, BytesList) -> ok | {error, Reason}`
[page 58] Log several items onto a disk log.
- `open(ArgL) -> OpenRet | DistOpenRet`
[page 59] Open a disk log file.
- `pid2name(Pid) -> {ok, Log} | undefined`
[page 62] Return the name of the disk log handled by a pid.
- `reopen(Log, File)`
[page 62] Reopen a disk log and save the old log.
- `reopen(Log, File, Head)`
[page 62] Reopen a disk log and save the old log.

- `breopen(Log, File, BHead) -> ok | {error, Reason}`
[page 62] Reopen a disk log and save the old log.
- `sync(Log) -> ok | {error, Reason}`
[page 62] Flush the contents of a disk log to the disk.
- `truncate(Log)`
[page 63] Truncate a disk log.
- `truncate(Log, Head)`
[page 63] Truncate a disk log.
- `btruncate(Log, BHead) -> ok | {error, Reason}`
[page 63] Truncate a disk log.
- `unblock(Log) -> ok | {error, Reason}`
[page 63] Unblock a disk log.

erl_boot_server

The following functions are exported:

- `start(Slaves) -> {ok, Pid} | {error, What}`
[page 64] Start the boot server.
- `start_link(Slaves) -> {ok, Pid} | {error, What}`
[page 64] Start the boot server and links the caller.
- `add_slave(Slave) -> ok | {error, What}`
[page 64] Add a slave to the list of allowed slaves.
- `delete_slave(Slave) -> ok | {error, What}`
[page 64] Delete a slave from the list of allowed slaves.
- `which_slaves() -> Slaves`
[page 65] Return the current list of allowed slave hosts.

erl_ddll

The following functions are exported:

- `start() -> {ok, Pid} | {error, Reason}`
[page 66] Start the server.
- `start_link() -> {ok, Pid} | {error, Reason}`
[page 66] Start the server and links it to the calling process.
- `stop() -> ok`
[page 66] Stop the server.
- `load_driver(Path, Name) -> ok | {error, ErrorDescriptor}`
[page 66] Load a driver.
- `unload_driver(Name) -> ok | {error, ErrorDescriptor}`
[page 66] Load a driver.
- `loaded_drivers() -> {ok, DriverList}`
[page 67] Load a driver.
- `format_error(ErrorDescriptor) -> string()`
[page 67] Format an error descriptor

erl_prim_loader

The following functions are exported:

- `start(Id, Loader, Hosts) -> {ok, Pid} | {error, What}`
[page 69] Start the Erlang low level loader.
- `get_file(File) -> {ok, Bin, FullName} | error`
[page 69] Get a file.
- `get_path() -> {ok, Path}`
[page 70] Get the path set in the loader.
- `set_path(Path) -> ok`
[page 70] Set the path of the loader.

erlang

The following functions are exported:

- `abs(Number)`
[page 73] Return an integer or float
- `erlang:append_element(Tuple, Term)`
[page 73] Return a new tuple
- `apply({Module, Function}, ArgumentList)`
[page 73] Return the result of applying Function in Module on ArgumentList
- `apply(Module, Function, ArgumentList)`
[page 73] Return the result of applying Function in Module on ArgumentList
- `atom_to_list Atom)`
[page 73] Return a list of integers
- `erlang:binary_to_float(Binary)`
[page 73] Return a float corresponding to the big-endian IEEE representation in Binary
- `binary_to_list(Binary)`
[page 74] Return a list of integers which correspond to the bytes of Binary
- `binary_to_list(Binary, Start, Stop)`
[page 74] Return a list of integers which correspond to the bytes of Binary
- `binary_to_term(Binary)`
[page 74] Return an Erlang term which is the result of decoding the binary Binary
- `erlang:bump_reductions(Reductions)`
[page 74] Increment the reduction counter for the current process
- `erlang:cancel_timer(Ref)`
[page 74] Cancel a timer
- `erlang:check_process_code(Pid, Module)`
[page 75] Return true if the process Pid is executing an old version of Module
- `concat_binary(ListOfBinaries)`
[page 75] >Concatenate a list of binaries
- `date()`
[page 75] Return the current date
- `erlang:delete_module(Module)`
[page 75] Move the current version of the code of Module to the old version

- `erlang:demonitor(Ref)`
[page 76] Turn off monitoring
- `erlang:disconnect_node(Node)`
[page 76] Force the disconnection of a node
- `erlang:display(Term)`
[page 76] Print a text representation `Term` on the standard output
- `element(N, Tuple)`
[page 76] Return the *N*th element (numbering from 1) of `Tuple`
- `erase()`
[page 76] Return and delete the process dictionary
- `erase(Key)`
[page 76] Return and delete the value associated with `Key` from the process dictionary
- `exit(Reason)`
[page 76] Stop the execution of the current process with the reason `Reason`
- `exit(Pid, Reason)`
[page 77] Send an EXIT message to the process `Pid`
- `erlang:fault(Reason)`
[page 77] Stop the execution of the current process with the reason `Reason`
- `erlang:fault(Reason, Args)`
[page 77] Stop the execution of the current process with the reason `Reason`
- `float(Number)`
[page 77] Return a float by converting `Number` to a float
- `erlang:float_to_binary(Float, Size)`
[page 78] Return a binary containing the big-endian IEEE representation of `Float`
- `float_to_list(Float)`
[page 78] >Returns a list of integers (ASCII codes) which corresponds to `Float`
- `erlang:fun_info(Fun)`
[page 78] Return a list containing information about the fun `Fun`
- `erlang:fun_info(Fun, Item)`
[page 78] Return information about the `Fun`
- `erlang:fun_to_list(Fun)`
[page 79] Return a textual representation of the fun `Fun`
- `erlang:function_exported(Module, Function, Arity)`
[page 79] Return true if a module is loaded and contains an exported function
- `erlang:garbage_collect()`
[page 79] Force an immediate garbage collection of the currently executing process
- `erlang:garbage_collect(Pid)`
[page 79] Force an immediate garbage collection of any process
- `get()`
[page 79] Return the process dictionary as a list of {`Key`, `Value`} tuples
- `get(Key)`
[page 79] Return the value associated with `Key` in the process dictionary
- `erlang:get_cookie()`
[page 79] >Return the "magic cookie" of the current node

- `get_keys(Value)`
[page 79] Return a list of keys which corresponds to `Value` in the process dictionary
- `group_leader()`
[page 80] Return the process identifier `Pid` of the group leader for the process evaluating the BIF
- `group_leader(Leader, Pid)`
[page 80] >Set the group leader of `Pid` to `Leader`
- `halt()`
[page 80] Halt the Erlang runtime system and indicate normal exit to the calling environment
- `halt(Status)`
[page 80] Halt the Erlang runtime system
- `erlang:hash(Term, Range)`
[page 80] Return a hash value for `Term`
- `hd(List)`
[page 81] Return the first item of a list
- `erlang:info(What)`
[page 81] Return system information on different 'topics' as binary data objects
- `integer_to_list(Integer)`
[page 81] Return a list of integers (ASCII codes) which correspond to `Integer`
- `is_alive()`
[page 81] Return the atom `true` if the current node is alive
- `erlang:is_builtin(Module, Function, Arity)`
[page 81] Return `true` if `Module:Function/Arity` is a BIF implemented in C
- `is_process_alive(Pid)`
[page 81] Return the atom `true` if the process is alive
- `length(List)`
[page 81] Return the length of a list
- `link(Pid)`
[page 81] Create a link to the process (or port) `Pid`
- `list_to_atom(CharIntegerList)`
[page 82] Return an atom whose text representation is the integers (Latin-1 codes) in `CharIntegerList`
- `list_to_binary(List)`
[page 82] Return a binary which is made from the integers and binaries in `List`
- `list_to_float(AsciiIntegerList)`
[page 82] Return a float whose text representation is the integers (ASCII-values) in `AsciiIntegerList`
- `list_to_integer(AsciiIntegerList)`
[page 82] Return an integer whose text representation is the integers (ASCII-values) in `AsciiIntegerList`
- `list_to_pid(AsciiIntegerList)`
[page 82] Return a `Pid` whose text representation is the integers (ASCII-values) in `AsciiIntegerList`
- `list_to_tuple(List)`
[page 82] Return a tuple which corresponds to `List`

- `erlang:load_module(Module, Binary)`
[page 83] Load object code
- `erlang:loaded()`
[page 83] Return a list of all loaded Erlang modules
- `erlang:localtime()`
[page 83] Return the current local date and time
- `erlang:localtime_to_universaltime(DateTime)`
[page 83] Convert local date and time in to Universal Time Coordinated (UTC)
- `make_ref()`
[page 84] Return an almost unique reference
- `erlang:make_tuple(Arity, InitialValue)`
[page 84] Return a new tuple of the given Arity
- `erlang:md5(Data) -> Digest`
[page 84] Compute an MD5 message digest from Data
- `erlang:md5_final(Context) -> Digest`
[page 84] Finishe the update of an MD5 Context and return the computed MD5 message digest
- `erlang:md5_init() -> Context`
[page 84] Create an MD5 context
- `erlang:md5_update(Context, Data) -> NewContext`
[page 84] Update an MD5 Context with Data, and return a NewContext
- `erlang:module_loaded(Module)`
[page 84] Return the atom true if the module contained in atom Module is loaded
- `erlang:monitor(Type, Item)`
[page 85] Start monitoring
- `monitor_node(Node, Flag)`
[page 85] Monitor the status of a node
- `node()`
[page 86] Return the name of the current node
- `node(Arg)`
[page 86] Return the node where Arg is located
- `nodes()`
[page 86] Return a list of all visible nodes in the system
- `nodes(Arg)`
[page 86] Return a list of nodes according to argument given
- `now()`
[page 87] Return the tuple {MegaSecs, Secs, Microsecs}, which is the elapsed time since 00:00 GMT
- `open_port(PortName, PortSettings)`
[page 87] Return a port identifier as the result of opening a new Erlang port
- `erlang:phash(Term, Range)`
[page 89] Portable hash function that will give the same hash for the same erlang term regardless of machine architecture and ERTS version
- `pid_to_list(Pid)`
[page 89] Return a list which corresponds to the process Pid
- `port_close(Port)`
[page 89] Close an open port

- `port_command(Port, Data)`
[page 90] Send data to a port
- `port_connect(Port, Pid)`
[page 90] Set the port owner (the connected port) to `Pid`
- `port_control(Port, Operation, Data)`
[page 90] Perform a synchronous control operation on a port.
- `port_call(Port, Operation, Data)`
[page 91] Synchronous call with term data.
- `erlang:port_info(Port, Item)`
[page 91] Return information about the port `Port`
- `erlang:port_to_list(Port)`
[page 91] Return a list which corresponds to the port identifier `Port`
- `erlang:ports()`
[page 92] Return a list of all ports on the current node
- `erlang:pre_loaded()`
[page 92] Return a list of pre-loaded modules
- `erlang:process_display(Pid, Type)`
[page 92] Write information about the local process `Pid` on standard error
- `process_flag(Flag, Option)`
[page 92] Set certain flags for the process which calls this function
- `process_flag(Pid, Flag, Option)`
[page 93] Set certain flags for the process `Pid`
- `process_info(Pid)`
[page 93] Return a long list which contains information about the process `Pid`
- `process_info(Pid, Item)`
[page 94] Return information about the process `Pid`
- `processes()`
[page 95] Return a list of all processes on the current node
- `erlang:purge_module(Module)`
[page 95] Removes old code for a module
- `put(Key, Value)`
[page 95] Add a new `Value` to the process dictionary and associate it with `Key`
- `erlang:read_timer(Ref)`
[page 95] Return the number of milliseconds remaining for a timer
- `erlang:ref_to_list(Ref)`
[page 95] >Return a list which corresponds to the reference `Ref`
- `register(Name, P)`
[page 96] Associate the name `Name` with the port or process identity `P`
- `registered()`
[page 96] Return a list of names which have been registered using `register/2`
- `erlang:resume_process(Pid)`
[page 96] Resume a suspended process
- `round(Number)`
[page 96] Return an integer by rounding the number `Number`
- `self()`
[page 96] Return the process identity of the calling process

- `erlang:send_after(Time, Pid, Msg)`
[page 96] Return a reference
- `erlang:set_cookie(Node, Cookie)`
[page 97] Set the "magic cookie" of Node to the atom Cookie
- `setelement(Index, Tuple, Value)`
[page 97] Return a tuple which is a copy of the argument Tuple
- `size(Item)`
[page 97] Return an integer which is the size of the argument Item
- `spawn(Fun)`
[page 97] Create a new Erlang process with a fun as entry point
- `spawn(Node, Fun)`
[page 97] Create a new Erlang process with a fun as entry point on a specified node
- `spawn(Module, Function, ArgumentList)`
[page 97] Create a new Erlang process with a specified function as entry point
- `spawn(Node, Module, Function, ArgumentList)`
[page 98] Create a new Erlang process with a specified function as entry point on a given node
- `spawn_link(Fun)`
[page 98] Create a new Erlang process with a fun as entry point and link to the new process
- `spawn_link(Node, Fun)`
[page 98] Create a new Erlang process with a fun as entry point on a specified node and link to the new process
- `spawn_link(Module, Function, ArgumentList)`
[page 98] Create a new Erlang process with a specified function as entry point and link to the new process
- `spawn_link(Node, Module, Function, ArgumentList)`
[page 98] Create a new Erlang process on a specified node with a function as the entry point and link to the new process
- `spawn_opt(Module, Function, ArgumentList, Options)`
[page 99] Create a new Erlang process with a function as entry point giving additional options
- `split_binary(Binary, Pos)`
[page 99] Return a tuple which contains two binaries which are the result of splitting Binary into two parts at position Pos
- `erlang:start_timer(Time, Pid, Msg)`
[page 100] Start a timer and return a reference to it
- `statistics(Type)`
[page 100] Return information about the system
- `erlang:suspend_process(Pid)`
[page 101] Suspend a process
- `erlang:system_flag(Flag, Value)`
[page 101] Set various system properties of the Erlang node
- `erlang:system_info(What)`
[page 101] Return system information on different 'topics' as binary data objects
- `term_to_binary(Term)`
[page 102] Return the encoded value of any Erlang term

- `term_to_binary(Term, Options)`
[page 102] Return the encoded value of any Erlang term
- `throw(Any)`
[page 102] Throw any exception
- `time()`
[page 102] Return the tuple {Hour, Minute, Second} of the current system time
- `tl(List)`
[page 103] Return a list stripped of its first element
- `erlang:trace(PidSpec, How, Flaglist)`
[page 103] Turn on or off the trace flags for a process or processes
- `erlang:trace_info(PidOrFunc, Item)`
[page 105] Return trace information about a process or exported function
- `erlang:trace_pattern(MFA, MatchSpec)`
[page 106] Set trace patterns for global call tracing
- `erlang:trace_pattern(MFA, MatchSpec, FlagList)`
[page 106] Set trace patterns for tracing of function calls
- `trunc(Number)`
[page 107] Return an integer by the truncation of Number
- `tuple_to_list(Tuple)`
[page 107] Return a list which corresponds to Tuple
- `erlang:universaltime()`
[page 108] Return the current date and time according to Universal Time Coordinated (UTC)
- `erlang:universaltime_to_localtime(DateTime)`
[page 108] Convert UTC date and time in to local date and time
- `unlink(Pid)`
[page 108] Remove a link, if there is one, from the calling process to another process
- `unregister(Name)`
[page 108] Remove the registered name for a port or process
- `whereis(Name)`
[page 108] Return the port or process corresponding to a registered name
- `erlang:yield()`
[page 108] Let other processes get a chance to execute

error_handler

The following functions are exported:

- `undefined_function(Module, Func, ArgList) -> term()`
[page 109] Called when an undefined function is encountered
- `undefined_lambda(Module, Fun, ArgList) -> term()`
[page 109] Called when an undefined lambda (fun) is encountered

error_logger

The following functions are exported:

- `start()` -> {ok, Pid} | {error, What}
[page 111] Start the error logger event manager.
- `start_link()` -> {ok, Pid} | {error, What}
[page 111] Start the error logger event manager.
- `error_report(Report)` -> ok
[page 111] Send a standard error report event to the error logger.
- `error_report(Type,Report)` -> ok
[page 112] Send a user defined error report type event.
- `info_report(Report)` -> ok
[page 112] Send an information report to the error logger.
- `info_report(Type,Report)` -> ok
[page 112] Send a user defined information report type event.
- `error_msg(Format)` -> ok
[page 113] Send an error event to the error logger.
- `error_msg(Format,Args)` -> ok
[page 113] Send an error event to the error logger.
- `format(Format,Args)` -> ok
[page 113] Send an error event to the error logger.
- `info_msg(Format)` -> ok
[page 113] Send an information event to the error logger.
- `info_msg(Format,Args)` -> ok
[page 113] Send an information event to the error logger.
- `tty(Flag)` -> ok
[page 113] Enable or disables error printouts to the tty.
- `logfile(Request)` -> ok | FileName | {error, What}
[page 113] Enable or disables error printouts to a file.
- `add_report_handler(Module)` -> ok | Other
[page 114] Add a new event handler to the error logger.
- `add_report_handler(Module,Args)` -> ok | Other
[page 114] Add a new event handler to the error logger.
- `delete_report_handler(Module)` -> Return | {error, What}
[page 114] Delete an error report handler.
- `swap_handler(ToHandler)` -> ok
[page 114] Swap from a primitive first handler to a standard event handler

file

The following functions are exported:

- `change_group(Filename, Gid)`
[page 116] Change owner for a file
- `change_owner(Filename, Uid)`
[page 116] Change owner of a file

- `change_owner(Filename, Uid, Gid)`
[page 116] Change owner for a file
- `change_time(Filename, Mtime)`
[page 116] Change the modification time for a file
- `change_time(Filename, Mtime, Atime)`
[page 116] Change the modification time for a file
- `close(IoDevice)`
[page 116] Close a file
- `consult(Filename)`
[page 117] Read Erlang terms from a file
- `copy(Source, Destination)`
[page 117] Copies file contents
- `copy(Source, Destination, ByteCount)`
[page 117] Copies file contents
- `del_dir(DirName)`
[page 117] Delete a directory
- `delete(Filename)`
[page 118] Delete a file
- `eval(Filename)`
[page 118] Evaluate expressions in a file
- `file_info(Filename)`
[page 118] Get information about a file
- `format_error(ErrorDescriptor)`
[page 119] Return an English description of an error term
- `get_cwd()`
[page 119] Get the current working directory
- `get_cwd(Drive)`
[page 119] Get the current working directory for the drive specified
- `ipread_s32bu_p32bu(IoDevice, Location, MaxSize)`
[page 119] Specialized indirect read function for Dets
- `list_dir(DirName)`
[page 120] List files in a directory
- `make_dir(DirName)`
[page 120] Make a directory
- `make_link(Existing, New)`
[page 120] Make a hard link to a file
- `make_symlink(Name1, Name2)`
[page 120] Make a symbolic link to a file or directory
- `open(Filename, ModeList)`
[page 121] Open a file
- `path_consult(Path, Filename)`
[page 122] Read Erlang terms from a file
- `path_eval(Path, Filename)`
[page 123] Evaluate expressions in a file
- `path_open(Path, Filename, Mode)`
[page 123] Open a file for access

- `position(IoDevice, Location)`
[page 123] Set position in a file
- `pread(IoDevice, [{Location, Number}, ...])`
[page 124] Read from a file at certain positions
- `pread(IoDevice, Location, Number)`
[page 124] Read from a file at a certain position
- `pwrite(IoDevice, [{Location, Bytes}, ...])`
[page 124] Write to a file at certain positions
- `pwrite(IoDevice, Location, Bytes)`
[page 124] Write to a file at a certain position
- `read(IoDevice, Number)`
[page 124] Read from a file
- `read_file(Filename)`
[page 124] Read a file
- `read_file_info(Filename)`
[page 125] Get information about a file
- `read_link(Linkname)`
[page 126] See what a link is pointing to
- `read_link_info(Filename)`
[page 126] Get information about a link or file
- `rename(Source, Destination)`
[page 126] Rename a file
- `set_cwd(DirName)`
[page 127] Set the current working directory
- `sync(IoDevice)`
[page 127] Synchronizes the in-memory state of a file with that on the physical medium
- `truncate(IoDevice)`
[page 127] Truncate a file
- `write(IoDevice, Bytes)`
[page 127] Write to a file
- `write_file(Filename, Binary)`
[page 127] Write a file
- `write_file_info(Filename, FileInfo)`
[page 128] Change file information

gen_tcp

The following functions are exported:

- `accept(ListenSocket) -> {ok, Socket} | {error, Reason}`
[page 134] Accept an incoming connection request on a listen socket.
- `accept(ListenSocket, Timeout) -> {ok, Socket} | {error, Reason}`
[page 134] Accept an incoming connection request on a listen socket.
- `close(Socket) -> ok | {error, Reason}`
[page 134] Close an TCP socket

- `connect(Address, Port, Options) -> {ok, Socket} | {error, Reason}`
[page 134] Connect to a TCP port.
- `connect(Address, Port, Options, Timeout) -> {ok, Socket} | {error, Reason}`
[page 134] Connect to a TCP port.
- `controlling_process(Socket, NewOwner) -> ok | {error, eperm}`
[page 135] Assign a new controlling process to a socket
- `listen(Port, Options) -> {ok, Socket} | {error, Reason}`
[page 135] Set up a socket which listen on Port
- `recv(Socket, Length) -> {ok, Packet} | {error, Reason}`
[page 135] Receive a packet from a passive socket
- `recv(Socket, Length, Timeout)`
[page 135] Receive a packet from a passive socket
- `send(Socket, Packet) -> ok | {error, Reason}`
[page 136] Send a packet

gen_udp

The following functions are exported:

- `close(Socket) -> ok | {error, Reason}`
[page 137] Close Socket.
- `controlling_process(Socket, NewOwner) ->`
[page 137] Change controlling process of a Socket.
- `open(Port) -> {ok, Socket} | {error, Reason}`
[page 137] Associate a UDP port number with the process calling it.
- `open(Port, Options) -> {ok, Socket} | {error, Reason}`
[page 137] Associate a UDP port number with the process calling it.
- `recv(Socket, Length) -> {ok, {Address, Port, Packet}} | {error, Reason}`
[page 138] Receive a packet from a passive socket
- `recv(Socket, Length, Timeout)`
[page 138] Receive a packet from a passive socket
- `send(S, Address, Port, Packet) -> ok | {error, Reason}`
[page 138] Send a packet to a specified Address and Port (from port associated with Id).

global

The following functions are exported:

- `del_lock(Id)`
[page 140] Delete the lock Id
- `del_lock(Id, Nodes) -> void()`
[page 140] Delete the lock Id
- `notify_all_name(Name, Pid1, Pid2) -> none`
[page 140] Name resolving function that notifies both Pids

- `random_exit_name(Name, Pid1, Pid2) -> Pid1 | Pid2`
[page 140] Name resolving function that kills one Pid
- `random_notify_name(Name, Pid1, Pid2) -> Pid1 | Pid2`
[page 140] Name resolving function that notifies one Pid
- `register_name(Name, Pid)`
[page 140] Globally registers Pid as Name
- `register_name(Name, Pid, Resolve) -> yes | no`
[page 140] Globally registers Pid as Name
- `registered_names() -> [Name]`
[page 141] Return all globally registered names
- `re_register_name(Name, Pid)`
[page 141] Atomically re-register Pid for Name
- `re_register_name(Name, Pid, Resolve) -> void()`
[page 141] Atomically re-register Pid for Name
- `send(Name, Msg) -> Pid`
[page 141] Send Msg to the global process Name
- `set_lock(Id)`
[page 141] Set a lock on the specified nodes
- `set_lock(Id, Nodes)`
[page 141] Set a lock on the specified nodes
- `set_lock(Id, Nodes, Retries) -> boolean()`
[page 141] Set a lock on the specified nodes
- `start()`
[page 142] Start the global name server
- `start_link() -> {ok, Pid} | {error, Reason}`
[page 142] Start the global name server
- `stop() -> void()`
[page 142] Stop the global name server
- `sync() -> void()`
[page 142] Synchronize the global name server
- `trans(Id, Fun)`
[page 142] Micro transaction facility
- `trans(Id, Fun, Nodes)`
[page 142] Micro transaction facility
- `trans(Id, Fun, Nodes, Retries) -> Res | aborted`
[page 142] Micro transaction facility
- `unregister_name(Name) -> void()`
[page 143] Unregister the global name Name
- `whereis_name(Name) -> Pid() | undefined`
[page 143] Return the Pid of the global process Name

global_group

The following functions are exported:

- `global_groups() -> {OwnGroupName, [OtherGroupName]} | undefined`
[page 145] Return the global group names

- `info()` -> [{state, State}, {own_group_name, atom()}, {own_group_nodes, [Node]}, {synced_nodes, [Node]}, {sync_error, [Node]}, {no_contact, [Node]}, {other_groups, Other_grps}, {monitoring, [pid()]}]
[page 145] Return the state of the global group process
- `monitor_nodes(Flag)` -> ok
[page 146] Subscription of node status for nodes in the immediate global group
- `own_nodes()` -> [Node] | {error, ErrorMsg}
[page 146] Return the global group names
- `registered_names({node, Node})` -> [Name] | {error, ErrorMsg}
[page 146] Return all globally registered names
- `registered_names({group, GlobalGroupName})` -> [Name]
[page 146] Return all globally registered names
- `send(Name, Msg)` -> Pid | {badarg, Msg} | {error, ErrorMsg}
[page 146] Send Msg to a registered process Name
- `send({node, Node}, Name, Msg)` -> Pid | {badarg, Msg} | {error, ErrorMsg}
[page 146] Send Msg to a registered process Name
- `send({group, GlobalGroupName}, Name, Msg)` -> Pid | {badarg, Msg} | {error, ErrorMsg}
[page 146] Send Msg to a registered process Name
- `sync()` -> ok
[page 146] Synchronize the immediate global group
- `whereis_name(Name)` -> Pid | undefined | {error, ErrorMsg}
[page 147] Return the Pid of the global process Name
- `whereis_name({node, Node}, Name)` -> Pid | undefined | {error, ErrorMsg}
[page 147] Return the Pid of the global process Name
- `whereis_name({group, GlobalGroupName}, Name)` -> Pid | undefined | {error, ErrorMsg}
[page 147] Return the Pid of the global process Name
- `start()`
[page 147] Start the global group server
- `start_link()` -> {ok, Pid} | {error, Reason}
[page 147] Start the global group server
- `stop()` -> void()
[page 147] Stop the global group server

heart

The following functions are exported:

- `start()` -> {ok, Pid} | ignore | {error, What}
[page 148] Start the heart program.
- `set_cmd(Cmd)` -> ok | {error, {bad_cmd, Cmd}}
[page 149] Set a temporary reboot command.
- `clear_cmd()` -> ok
[page 149] Clear the temporary boot command.

inet

The following functions are exported:

- `format_error(Tag)`
[page 150] Return a diagnostic error string.
- `gethostbyaddr(Address) -> {ok, Hostent} | {error, Reason}`
[page 150] Return a hostent record for the host with the given address
- `gethostbyname(Name) -> {ok, Hostent} | {error, Reason}`
[page 150] Return a hostent record for the host with the given name
- `gethostbyname(Name, Family) -> {ok, Hostent} | {error, Reason}`
[page 151] Return a hostent record for the host with the given name
- `gethostname() -> {ok, Name} | {error, Reason}`
[page 151] Return the local hostname
- `sockname(Socket) -> {ok, {IP, Port}} | {error, Reason}`
[page 151] Return the local address and port number for a socket.
- `peername(Socket) -> {ok, {Address, Port}} | {error, Reason}`
[page 151] Return the address and port for the other end of a connection.
- `port(Socket) -> {ok, Number}`
[page 151] Return the local port number for a socket.
- `close(Socket) -> ok`
[page 151] Close a socket of any type
- `getaddr(IP, inet) -> {ok, {A1,A2,A3,A4}} | {error, Reason}`
[page 152] Return the IP-address for IP
- `setopts(Socket, Options) -> ok | {error, Reason}`
[page 152] Set one or more options for a socket.

init

The following functions are exported:

- `boot(BootArgs) -> void()`
[page 157] Start the Erlang runtime system.
- `get_arguments() -> Flags`
[page 157] Get all flag arguments.
- `get_argument(Flag) -> {ok, Values} | error`
[page 158] Get values associated with an argument.
- `get_args() -> [Arg]`
[page 158] Get all (non-flag) arguments.
- `get_plain_arguments() -> [Arg]`
[page 158] Get all (non-flag) arguments.
- `restart() -> void()`
[page 158] Restart the running Erlang node
- `reboot() -> void()`
[page 158] Take down an Erlang node smoothly
- `stop() -> void()`
[page 158] Take down an Erlang node smoothly<

- `get_status()` -> {InternalStatus, ProvidedStatus}
[page 159] Get status information during system start.
- `script_id()` -> Id
[page 159] Get the identity of the used boot script.

net_adm

The following functions are exported:

- `host_file()`
[page 162] Read the `.hosts.erlang` file
- `dns_hostname(Host)`
[page 162] Call `epmd` for the fully qualified name (DNS) of `Host`
- `localhost()`
[page 162] Return the fully qualified name of the local host
- `names(), names(Host)`
[page 162] Return {ok, List} or {error, Reason}
- `ping(Node)`
[page 162] Set up a connection to a node
- `world (), world (verbose)`
[page 162] Run `epmd - names` on all hosts which are specified in the Erlang host file
- `worldlist (Hostlist), worldlist (Hostlist, verbose)`
[page 162] Run `epmd - names` on all hosts which are specified in the Erlang host file

net_kernel

The following functions are exported:

- `kernel_apply(M, F, A)`
[page 164] Provide another net kernel
- `monitor_nodes(Flag)`
[page 164] Receive copies of messages that the net kernel receives from the runtime system
- `allow(NodeList)`
[page 165] Limit access to a node from a specific number of named nodes
- `connect_node(Node)`
[page 165] Establish a connection to a node

OS

The following functions are exported:

- `cmd(Command) -> string()`
[page 166] Execute Command in a command shell of the target OS.
- `find_executable(Name) -> Filename | false`
[page 166] Return the absolute filename of a program.
- `find_executable(Name, Path) -> Filename | false`
[page 166] Return the absolute filename of a program.
- `getenv() -> List`
[page 166] Return a list of all environment variables.
- `getenv(VarName) -> Value | false`
[page 167] Return the Value of the environment variable VarName.
- `getpid() -> Value`
[page 167] Return the process identifier of the emulator process as a string.
- `putenv(VarName, Value) -> true`
[page 167] Set a new Value for the environment variable VarName.
- `type() -> {Osfamily, Osname} | Osfamily`
[page 167] Return the Osfamily and, in some cases, Osname of the current operating system.
- `version() -> {Major, Minor, Release} | VersionString`
[page 167] Return the Operating System version.

pg2

The following functions are exported:

- `create(Name) -> void()`
[page 169] Create a new, empty process group
- `delete(Name) -> void()`
[page 169] Delete a process group
- `get_closest_pid(Name) -> Pid | {error, Reason}`
[page 169] Common dispatch function
- `get_members(Name) -> [Pid] | {error, Reason}`
[page 170] Return all processes in a group
- `get_local_members(Name) -> [Pid] | {error, Reason}`
[page 170] Return all local processes in a group
- `join(Name, Pid) -> ok | {error, Reason}`
[page 170] Join a process to a group
- `leave(Name, Pid) -> ok | {error, Reason}`
[page 170] Make a process leave a group
- `which_groups() -> [Name]`
[page 170] Return a list of all known groups
- `start()`
[page 170] Start the pg2 server
- `start_link() -> {ok, Pid} | {error, Reason}`
[page 170] Start the pg2 server

rpc

The following functions are exported:

- `start()`
[page 171] Start the rpc server
- `stop()`
[page 171] Stop the rpc server
- `call(Node, Module, Function, Args)`
[page 171] Evaluate a function call on a node
- `cast(Node, Module, Function, Args)`
[page 171] Run a function on a node ignoring the result
- `block_call(Node, Mod, Fun, Args)`
[page 171] Evaluate a function call on a remote node in the RPC servers context
- `server_call(Node, Name, ReplyWrapper, Msg)`
[page 172] Interact with a server on a node
- `abcast(Name, Mess)`
[page 172] Broadcast a message asynchronously to a registered process on all nodes
- `abcast(Nodes, Name, Mess)`
[page 172] Broadcast a message asynchronously to a registered process on specific nodes
- `sbcast(Name, Msg)`
[page 172] Broadcast to all nodes synchronously and return a list of the nodes which have a registered server
- `sbcast(Nodes, Name, Msg)`
[page 172] Broadcast to specific nodes synchronously and return a list of the nodes which have a registered server
- `eval_everywhere(Mod, Fun, Args)`
[page 172] Run a function call on all nodes
- `eval_everywhere(Nodes, Mod, Fun, Args)`
[page 172] Run a function call on a specified set of nodes
- `multicall(M, F, A)`
[page 172] Evaluate a function call on all nodes
- `multicall(Nodes, M, F, A)`
[page 173] Evaluate a function call on a specific set of nodes
- `multi_server_call(Name, Msg)`
[page 173] Send a message to servers on all nodes and collect the answers
- `multi_server_call(Nodes, Name, Msg)`
[page 173] Send a message to servers on a specific set of nodes and collect the answers
- `safe_multi_server_call(Name, Msg)`
[page 173] Send a message to servers on all nodes and collect the answers safely
- `safe_multi_server_call(Nodes, Name, Msg)`
[page 174] >Send a message to servers on a specific set of nodes and collect the answers safely
- `async_call(Node, Mod, Fun, Args)`
[page 174] Evaluate a function asynchronously on a node and return a key which can be used at a later stage to collect results

- `yield(Key)`
[page 174] Deliver the promised answer from a `async_call` operation
- `nb_yield(Key, Timeout)`
[page 174] Deliver the promised answer from a `async_call` operation (non-blocking)
- `nb_yield(Key)`
[page 174] Deliver the promised answer from a `async_call` operation (non-blocking)
- `parallel_eval(ListOfTuples)`
[page 174] Evaluate several function calls on all nodes in parallel
- `pmap({M, F}, Extraargs, List)`
[page 174] Execute `lists:map/3` in parallel on all nodes
- `pinfo(Pid)`
[page 174] Get process info for any process
- `pinfo(Pid, Item)`
[page 174] Get process info for any process

seq_trace

The following functions are exported:

- `set_token(Component, ComponentValue) -> {Component, PreviousValue}`
[page 175] Set the individual Component of the trace token.
- `set_token(Token) -> PreviousToken`
[page 176] Set the trace token to Value.
- `get_token(Component) -> {Component, ComponentValue}`
[page 176] Return the ComponentValue of the trace token component Component.
- `get_token() -> TraceToken`
[page 176] Return the value of the trace token.
- `print(TraceInfo) -> void`
[page 176] Put the Erlang term TraceInfo into the sequential trace output.
- `print(Label, TraceInfo) -> void`
[page 177] Put the Erlang term TraceInfo into the sequential trace output.
- `reset_trace() -> void`
[page 177] Stop all sequential tracing on the Erlang node.
- `set_system_tracer(ProcessOrPortId) -> PreviousId`
[page 177] Set the system tracer.
- `get_system_tracer() -> pid() | port() | false`
[page 177] Return the pid() or port() of the current system tracer.

user

The following functions are exported:

- `start() -> void()`
[page 183] Start the standard I/O system.

wrap_log_reader

The following functions are exported:

- `chunk(Continuation)`
[page 184] Read a chunk of objects written to a wrap log.
- `chunk(Continuation, N) -> {Continuation2, Terms} | {Continuation2, Terms, Badbytes} | {Continuation2, eof} | {error, Reason}`
[page 184] Read a chunk of objects written to a wrap log.
- `close(Continuation) -> ok`
[page 185] Close a log
- `open(Filename) -> OpenRet`
[page 185] Open a log file
- `open(Filename, N) -> OpenRet`
[page 185] Open a log file

app

No functions are exported.

config

No functions are exported.

kernel

Application

The Kernel application is the first application started, and it is one of two mandatory applications. The Kernel application contains the following services:

- `application_controller`
- `auth`
- `code`
- `disk_log`
- `erl_boot_server`
- `erl_ddll`
- `error_logger`
- `file`
- `global_group`
- `global_name_server`
- `net_kernel`
- `os`
- `rpc`
- `pg2`
- `timer`
- `user`

It is possible to synchronize a set of Erlang nodes. One can specify for a node to wait a specified amount of time for other nodes to become alive.

Error Logger Event Handlers

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

Configuration

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

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

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

These parameters are described in `application(3)`.

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

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

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

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

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

`error_logger = Value <optional>` Value is one of:

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

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

`false` No error logger handler is installed.

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

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

These parameters are described in `global_group(3)`.

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

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

`net_ticktime = TickTime <optional>` Specifies the `net_kernel` tick time.

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

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


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

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

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

Note: Normally, a terminating node is detected immediately.

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

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

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

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

The default value is false.

`start_dist_ac = true | false <optional>` Starts the `dist_ac` server if the parameter is true (see `application(3)`). This parameter should be set to true for systems that use distributed applications.

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

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

The default value is false.

`boot_server_slaves = [SlaveIP] <optional>` If the `start_boot_server` configuration parameter is true, this parameter can be used to initialize `boot_server` with a list of slave IP addresses. `SlaveIP = string() | atom | {integer(),integer(),integer(),integer()}` where $0 \leq \text{integer()} \leq 255$.

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

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

The default value is [].

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

The default value is false.

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

The default value is false.

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

The default value is `false`.

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

The default value is 0.

See Also

`application(3)`, `auth(3)`, `code(3)`, `disk_log(3)`, `erl_ddll(3)`, `erl_boot_server(3)`, `error_logger(3)`, `file(3)`, `global(3)`, `global_group(3)`, `net_kernel(3)`, `pg2(3)`, `rpc(3)`, `timer(3)`, `user(3)`

application

Erlang Module

In OTP, *application* is an abstract term used to define a system component consisting of resources such as modules and processes. This module contains functions for controlling applications (for example starting and stopping applications), and functions to access information about applications (for example configuration parameters).

An application is defined by an *application specification*. The specification is normally located in an *application resource file* called `Application.app`, where `Application` is the name of the application. Refer to `app(4)` for more information about the application specification.

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

An application can include other applications. An application which is not included by any other application is called a *primary application*. An included application should be placed underneath a supervisor in the including application. This means that when running, an included application – although conceptually a self-contained system component – is in practice part of a primary application. A process in an included application will consider itself belonging to the primary application. Unless otherwise stated, the functions in this module are not directly applicable to included applications. Refer to *OTP Design Principles* for more information about applications and behaviours.

Application Handling

An *application controller* is automatically started at every Erlang node. When starting an application, the application controller starts an *application master* which monitors that application.

An application is said to be local or distributed. A local application is started at any node and, if the node goes down, the application terminates. A distributed application is started at one of a specified list of nodes and, if the node goes down, the application will be restarted automatically at one of the other nodes in the list (*failover*). If the first node comes up again, the application will be moved back to it (*takeover*). See `takeover/2` below for more information about takeovers.

Local applications are controlled by the application controller. Distributed applications are controlled by the *distributed application controller* (`dist_ac`). The distributed application controller is not started by default; systems that use distributed applications must make sure that Kernel is configured so that `dist_ac` is started and the nodes are synchronized. Refer to `kernel(6)` for more information.

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, which may be an included 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, which may be an included 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, which may be an included 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, which may be an included 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(ApplDescr) -> ok | {error, Reason}
```

```
load(ApplDescr, Distributed) -> ok | {error, Reason}
```

Types:

- ApplDescr = Application | {application, Application, AppSpec}
- Application = atom()
- 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 can also be given by the tuple {application, Application, AppSpec}. In this case the application controller will load the data in AppSpec instead.

AppSpec should be a list of application specification key-value tuples. Invalid keys are ignored and default values are used for any omitted keys. Refer to app(4) for information about valid key-value tuples.

If the `Distributed` argument is present, the application will be distributed. The argument overrides the value for the application in the Kernel configuration parameter `distributed`.

In `Distributed = {Application, [Time,]Nodes}`, `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 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, local 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 local 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 it is configured.

The function does not return until the application is started, stopped or successfully moved to another node. However, in some cases where permission is set to `true` the function may return `ok` even though the application itself has not started. This is true when an application cannot start because it has dependencies to other applications

which have not yet been started. When they have been started, `Application` will be started as well.

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

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

Types:

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

Sets the value of the configuration parameter `Par` for `Application`, which may be an included application.

Warning:

Use this function only if you know what you are doing, that is, on your own applications. It is very application and configuration parameter dependent when and how often the value is read by the application, and careless use of this function may put the application in a wierd, 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 similar to 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 `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
```

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.

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

Types:

- `Application = atom()`
- `Reason = term()`
Test of d-tag

Stops `Application`. All processes in the application tree are terminated, including included applications, and also all processes with the same group leader as the application. The application will still be 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.

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

Types:

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

Returns a list with information about the applications which are currently running. `Application` is the application name. `Description` and `Vsn` are the values of its `description` and `vsn` application specification keys, respectively.

CALLBACK MODULE

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

Exports

```
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.

```
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 provided in the application specification for the 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 called during startup of an application for each phase defined for the application specification key `start_phases` = `[{Phase,PhaseArgs}]`.

If it is desired that all or a subset of the start phases are called for any included applications as well, use `application_starter` and define the application specification key `mod` as:

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

Note that `application_starter` is not recursive, but has to be used for every application with included applications, if there are several levels of included applications.

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`.

SEE ALSO

`kernel(6)`, `app(4)`

auth

Erlang Module

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

Whenever a message is transferred from one node to another, it is accompanied by the `Magic Cookie` of the receiving node. For example, a message transferred from node A to node B is accompanied by what node A believes to be the `Magic Cookie` of node B.

When the message arrives at node B, the runtime system immediately checks that the accompanying cookie is the right one. If it is, the message is passed on in the normal way. If it is not, the message is transformed into a `badcookie` message, which is sent to the system process `net_kernel`. By default, the `net_kernel` process passes the message to the registered process `auth`, which is then responsible for taking the appropriate action for the unauthorized message. In the standard system, the default action is to shut down connection to that node.

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

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

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

To communicate with another node, the magic cookie of that node must be known. The BIF `erlang:set_cookie(Node, Cookie)` sets the cookie for `Node` to `Cookie`. From then on, all messages will be accompanied by the cookie `Cookie`. If the cookie is not correct when messages arrive at `Node`, they are sent to the `auth` server at `Node`. The call `erlang:set_cookie(node(), CookieAtom)` will set the current cookie to `CookieAtom`. It will, however, also set the cookie of all other unknown nodes to `CookieAtom`. In the case of the default `auth` server, this is the first thing done when the system starts. The default then, is to assume that all nodes which communicate have the same cookie. In the case of a single user on a single file system, this is indeed true and no further action is required. The original cookie can also be fetched by the BIF `erlang:get_cookie()`.

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

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

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

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

Exports

`start()`

Starts the auth server.

`stop()`

Stops the auth server.

`open(Name)`

This function opens up the server with the name `Name`. If, for example, node `N` is run with the cookie `C`, it is impossible for other nodes with other cookies to communicate with node `N`. The call `open/1` opens the server with the registered name `Name` so it can be accessed by any other node, irrespective of cookie. The call must be executed on both nodes to have any effect. All messages to the server must have the form `Name ! Msg` and all replies from the server `{Name, Reply}`, or `{Name, Node, Reply}`. With this feature, it is possible to perform specific tasks on publicly announced Erlang network servers.

`is_auth(Node)`

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

`exists(Node)`

Returns `yes` if `Node` exists, otherwise `no`.

`cookie()`

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

`node_cookie(Node, Cookie)`

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

`node_cookie([Node, Cookie])`

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

`cookie([Cookie])`

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

code

Erlang Module

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

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

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

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

Exports

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

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

Types:

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

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

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

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

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

Types:

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

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

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

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

Types:

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

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

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

Types:

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

Returns the current path.

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

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

Types:

- `Dir = string()`
- `What = bad_directory`

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

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

Types:

- `Dir = string()`
- `What = bad_directory`

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

```
add_paths(DirList) -> ok
```

```
add_pathsz(DirList) -> ok
```

Types:

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

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

```
add_pathsa(DirList) -> ok
```

Types:

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

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

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

Types:

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

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

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

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

Types:

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

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

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

Types:

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

This function tries to load the Erlang module `Module`, using the current path. It looks for the object code file which has a suffix that corresponds to the Erlang machine used, for example `Module.beam`. The loading fails if the module name found in the object code differs from the name `Module`. `load_binary/3` must be used to load object code with a module name that is different from the file name.

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

Types:

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

This function does the same as `load_file(Module)`, but `File` is either an absolute file name, or a relative file name. The current path is not searched. It returns a value in the same way as `load_file(Module)`. Note that `File` should not contain an extension (`".beam"`); `load_abs/1` adds the correct extension itself.

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

Types:

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

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

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

`delete(Module) -> true | false`

Types:

- Module = atom()

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

`purge(Module) -> true | false`

Types:

- Module = atom()

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

`soft_purge(Module) -> true | false`

Types:

- Module = atom()

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

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

Types:

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

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

`all_loaded() -> [LoadMod]`

Types:

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

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

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

Types:

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

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

`stop() -> stopped`

Stops the code server.

`root_dir() -> RootDir`

Types:

- `RootDir = string()`

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

`lib_dir() -> LibDir`

Types:

- `LibDir = string()`

Returns the library directory.

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

Types:

- `Name = atom()`
- `LibDir = string()`

- What = bad_name

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

compiler_dir() -> CompDir

Types:

- CompDir = string()

This function returns the compiler directory.

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

Types:

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

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

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

Types:

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

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

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

objfile_extension() -> Ext

Types:

- Ext = string()

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

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

Types:

- Dir = string()

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

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

Types:

- `Dir = string()`

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

```
which(Module) -> WhichFile
```

Types:

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

If the module is not loaded already, this function returns the directory path to the first file name in the search path of the code server which contains the object code for `Module`. If the module is loaded, it returns the directory path to the file name which contains the loaded object code. If the module is pre-loaded, `preloaded` is returned. If the module is Cover compiled, `cover_compiled` is returned. `non_existing` is returned if the module cannot be found.

```
clash() -> ok
```

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

Notes

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

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

disk_log

Erlang Module

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

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

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 60]. 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 *distributed disk log* if requests made to any one of the logs is 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. 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`, `chunk_step` and `lclose`. An open disk log that is not a distributed disk log is said to be a *local disk log*. A local disk log is accessible only from the node where the disk log process runs, whereas a distributed disk log is accessible from all nodes in the Erlang system, with exception for those nodes where a local disk log with the same name as the distributed disk log exists. All processes on nodes that have access to a local or distributed disk log can log items or otherwise change, inspect or close the log.

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

Note:

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

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

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

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

Exports

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

Types:

- LocalLog = DistributedLog = term()

The `accessible_logs/0` function returns the names of the disk logs accessible on the current node. The first list contains local disk logs, and the second list contains distributed disk logs.

```
alog(Log, Term) -> ok | {error, Reason}
```

```
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) -> ok | {error, Reason}
```

```
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`, `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}
```

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} | {not_internal_wrap, Log} | {corrupt_log_file, FileName} | {file_error, FileName, FileError}

The `chunk/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 8 kilobyte chunks from the file.

The first time `chunk` 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 8 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` 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} | {file_error, FileName, FileError}`

The function `chunk_step` can be used in conjunction with `chunk/2,3` to search through an internally formatted wrap log. It takes as argument a continuation as returned by `chunk/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 is not terminated as long as there are owners or users of the log. It should be stressed that each and every owner must close the log, possibly by terminating, and that any other process - not only the processes that have opened the log anonymously - can decrement the users counter by closing the log. Attempts to close a log by a process that is not an owner are simply ignored if there are no users.

If the log is blocked by the closing process, the log is also unblocked.

```
format_error(Error) -> character_list()
```

Given the error returned by any function in this module, the function `format_error` returns a descriptive string of the error in English. For file errors, the function `format_error/1` in the `file` module is called.

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

Types:

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

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

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

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

The `info/1` function returns a list of `{Tag, Value}` pairs describing the log. If there is a disk log process running on the current node, that log is used as source of information, otherwise an individual distributed log on some other node is chosen, if such a log exists.

The following pairs are returned for all logs:

- `{name, Log}`, where `Log` is the name of the log as given by the `open/1` option `name`.
- `{file, File}`. For halt logs `File` is the filename, and for wrap logs `File` is the base name.
- `{type, Type}`, where `Type` is the type of the log as given by the `open/1` option `type`.
- `{format, Format}`, where `Format` is the format of the log as given by the `open/1` option `format`.
- `{size, Size}`, where `Size` is the size of the log as given by the `open/1` option `size`, or the size set by `change_size/2`. The value set by `change_size/2` is reflected immediately.
- `{mode, Mode}`, where `Mode` is the mode of the log as given by the `open/1` option `mode`.

- {owners, [{pid(), Notify}]} where Notify is the value set by the open/1 option notify or the function change_notify/3 for the owners of the log.
- {users, Users} where Users is the number of anonymous users of the log, see the open/1 option linkto [page 60].
- {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_head/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 and chunk_step/3 functions do not affect any value returned by info/1.

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

```
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 56].

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

```
log(Log, Term) -> ok | {error, Reason}
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) -> ok | {error, Reason}
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. In case of using one of the functions that append a list of items to a log fills up several files, only one message is sent. `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_head/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 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)`, `pg2(3)`, `wrap_log_reader [page 184](3)`

erl_boot_server

Erlang Module

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

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

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

Exports

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

Types:

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

Starts the boot server. `Slaves` is a list of IP addresses for hosts which are allowed to use this server as a boot server.

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

Types:

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

Starts the boot server and links to the caller. This function is used to start the server if it is included in a supervision tree.

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

Types:

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

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

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

Types:

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

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

`which_slaves()` -> Slaves

Types:

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

Returns the current list of allowed slave hosts.

SEE ALSO

`init(3)`, `erl_prim_loader(3)`

erl_ddll

Erlang Module

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

Exports

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

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

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

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

`stop() -> ok`

Stops `ddll_server`.

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

Types:

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

Loads and links the dynamic driver `Name`. `Name` must be sharable object/dynamic library. Two drivers with different `Paths` cannot be loaded under the same name. The number of dynamically loadable drivers are limited by the size of `driver_tab` in `config.c`.

If the server is not started the caller will crash.

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

Types:

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

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

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

If the server is not started the caller will crash.

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

Types:

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

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

If the server is not started the caller will crash.

```
format_error(ErrorDescriptor) -> string()
```

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

Differences Between Statically Linked-in Drivers and Dynamically Loaded Drivers

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

The driver_entry struct has two new members: finish and handle.

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

The member handle contains a pointer obtained from the operating system when the driver was loaded. Without this, the driver cannot be unloaded!

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

Example:

```
#include <stdio.h>
#include "driver.h"
static long my_start();
static int my_stop(), my_read();
static struct driver_entry my_driver_entry;
/*
 * Initialize and return a driver entry struct
 */
struct driver_entry *driver_init(void *handle)
{
    my_driver_entry.init = null_func;      /* Not used */
    my_driver_entry.start = my_start;
    my_driver_entry.stop = my_stop;
    my_driver_entry.output = my_read;
    my_driver_entry.ready_input = null_func;
    my_driver_entry.ready_output = null_func;
    my_driver_entry.driver_name = "my_driver";
    my_driver_entry.finish = null_func;
```

```
    my_driver_entry.handle = handle; /* MUST set this!!! */  
    return &my_driver_entry;  
}
```

config.c

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

Compiling Your Driver

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

The include file `driver.h` is found in the `usr/include` directory of the Erlang installation.

erl_prim_loader

Erlang Module

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

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

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

Exports

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

Types:

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

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

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

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

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

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

Types:

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

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

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

Types:

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

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

```
set_path(Path) -> ok
```

Types:

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

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

Protocol

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

Function	Send	Receive
get_file	[102 FileName]	[121 BinaryFile] (on success) [122] (failure)
stop	eof	terminate

Command Line Flags

The `erl_prim_loader` module interprets the following flags:

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

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

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

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

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

SEE ALSO

`init(3)`, `erl_boot_server(3)`

erlang

Erlang Module

By convention, Built In Functions (BIFs) are seen as being in the module `erlang`. Thus, both the calls `atom_to_list(Erlang)` and `erlang:atom_to_list(Erlang)` are identical.

BIFs may fail for a variety of reasons. All BIFs fail if they are called with arguments of an incorrect type. For example, `atom_to_list/1` will fail if it is called with an argument which is not an atom. If this type of failure is not within the scope of a catch (and the BIF is not called within a guard; see below), the process making the call will exit, and an EXIT signal with the associated reason `badarg` will be sent to all linked processes. The other reasons that may make BIFs fail are described in connection with the description of each individual BIF.

A few BIFs may be used in guard tests, for example:

```
tuple_5(Something) when size(Something) == 5 ->
    is_tuple_size_5;
tuple_5(_) ->
    is_something_else.
```

Here the BIF `size/1` is used in a guard. If `size/1` is called with a tuple, it will return the size of the tuple (i.e., how many elements there are in the tuple). In the above example, `size/1` is used in a guard which tests if its argument `Something` is a tuple and, if it is, whether it is of size 5. In this case, calling `size` with an argument other than a tuple will cause the guard to fail and execution will continue with the next clause. Suppose `tuple_5/1` is written as follows:

```
tuple_5(Something) ->
    case size(Something) of
        5 -> is_tuple_size_5;
        _ -> is_something_else
    end.
```

In this case, `size/1` is not in a guard. If `Something` is not a tuple, `size/1` will fail and cause the process to exit with the associated reason `badarg` (see above).

Some of the BIFs in this chapter are optional in Erlang implementations, and not all implementations will include them.

The following descriptions indicate which BIFs can be used in guards and which BIFs are optional.

Exports

`abs(Number)`

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

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

This BIF is allowed in guard tests.

Failure: badarg if the argument is not an integer or a float.

`erlang:append_element(Tuple, Term)`

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

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

`apply({Module, Function}, ArgumentList)`

This is equivalent to `apply(Module, Function, ArgumentList)`.

`apply(Module, Function, ArgumentList)`

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

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

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

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

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

`atom_to_list Atom`

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

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

Failure: badarg if the argument is not an atom.

`erlang:binary_to_float(Binary)`

Returns a float corresponding to the big-endian IEEE representation in Binary. The size of Binary must be 4 or 8 bytes.

Warning:

This is an internal BIF, only to be used by OTP code.

Failure: `badarg` if the argument is not a binary or not the representation of a number.

`binary_to_list(Binary)`

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

`binary_to_list(Binary, Start, Stop)`

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

`binary_to_term(Binary)`

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

`erlang:bump_reductions(Reductions)`

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

Warning:

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

`erlang:cancel_timer(Ref)`

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

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

```

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

```

Failure: badarg if Ref is not a reference.

`erlang:check_process_code(Pid, Module)`

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

```

> erlang:check_process_code(Pid, lists).
false

```

This is an optional BIF.

Failure: badarg, if the process argument is not a `Pid`, or the module argument is not an atom.

`concat_binary(ListOfBinaries)`

Concatenates a list of binaries `ListOfBinaries` into one binary.

`date()`

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

```

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

```

`erlang:delete_module(Module)`

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

```

> delete_module(test).
true

```

This is an optional BIF.

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

Warning:

In normal Erlang implementations code handling - which includes loading, deleting, and replacing modules - is performed in the module code. This BIF is intended for use with the implementation of the module code and should not be used elsewhere.

`erlang:demonitor(Ref)`

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

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

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

`erlang:disconnect_node(Node)`

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

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

`erlang:display(Term)`

Prints a text representation `Term` on the standard output. Useful for debugging (especially startup problems) and strongly discouraged for other purposes.

`element(N, Tuple)`

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

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

Failure: `badarg` if $N < 1$, or $N > \text{size}(\text{Tuple})$, or if the argument `Tuple` is not a tuple. Allowed in guard tests.

`erase()`

Returns the process dictionary and deletes it.

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

`erase(Key)`

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

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

`exit(Reason)`

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


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

`exit(Pid, Reason)`

Sends an EXIT message to the process `Pid`. Returns `true`.

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

Note:

The above is not necessarily the same as:

```
Pid ! {'EXIT', self(), goodbye}
```

The above two alternatives are the same if the process with the process identity `Pid` is trapping exits. However, if `Pid` is not trapping exits, the `Pid` itself will exit and propagate EXIT signals in turn to its linked processes.

If the reason is the atom `kill`, for example `exit(Pid, kill)`, an untrappable EXIT signal will be sent to the process `Pid`. In other words, the process `Pid` will be unconditionally killed.

Returns `true`.

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

`erlang:fault(Reason)`

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

`erlang:fault(Reason, Args)`

Stops the execution of the current process with the reason `Reason`, where `Reason` is any Erlang term. The actual EXIT term will be `{Reason, Where}`, where `Where` is a list of the functions most recently called (the current function first). The `Args` is expected to be the 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.

`float(Number)`

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

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

Note:

`float/1` is allowed in guard tests, but it tests whether the argument is a float or not.

```
-module(t).

f(F) when float(F) -> float;
f(F) -> not_a_float.

1> t:f(1).
not_a_float
2> t:f(1.0).
float
3>
```

Failure: `badarg` if the argument is not a float or an integer.

`erlang:float_to_binary(Float, Size)`

Returns a binary containing the big-endian IEEE representation of `Float`. `Size` is the size in bits, and must be either 32 or 64.

Warning:

This is an internal BIF, only to be used by OTP code.

Failure: `badarg` if the argument is not a float.

`float_to_list(Float)`

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

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

Failure: `badarg` if the argument is not a float.

`erlang:fun_info(Fun)`

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

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

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

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

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

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

`erlang:fun_info(Fun, Item)`

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

`erlang:fun_to_list(Fun)`

Returns a textual representation of the fun Fun.

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

Returns true if the module Module is loaded and it contains an exported function Function/Arity; otherwise returns false. Returns false for any BIF (functions implemented in C rather than in Erlang).

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

`erlang:garbage_collect()`

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

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

`erlang:garbage_collect(Pid)`

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

`get()`

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

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

`get(Key)`

Returns the value associated with Key in the process dictionary, and undefined if no value is associated with Key. Key can be any Erlang term.

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

`erlang:get_cookie()`

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

`get_keys(Value)`

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

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

`group_leader()`

Every process is a member of some process group and all groups have a leader.

This BIF returns the process identifier `Pid` of the group leader for the process which evaluates the BIF. When a process is spawned, the group leader of the spawned process is the same as that of the process which spawned it. Initially, at system start-up, `init` is both its own group leader and the group leader of all processes.

`group_leader(Leader, Pid)`

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

`halt()`

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

```
> halt().  
unix_prompt%
```

`halt(Status)`

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

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

`erlang:hash(Term, Range)`

Returns a hash value for `Term` within the range `1..Range`. The allowed range is `1..227-1`.

Warning:

This BIF is deprecated as the hash value may differ on different architectures. Also the hash values for integer terms larger than `227` 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 the BIF `erlang:phash/2` instead.

`hd(List)`

Returns the first item of `List`.

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

Allowed in guard tests.

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

`erlang:info(What)`

This BIF is now equivalent to `erlang:system_info/1` [page 101].

`integer_to_list(Integer)`

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

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

Failure: `badarg` if the argument is not an integer.

`is_alive()`

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

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

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

`is_process_alive(Pid)`

`Pid` must refer to a process on the current node. Returns the atom `true` if the process is alive, i.e., has not exited. Otherwise, it returns the atom `false`. This is the preferred way to check whether a process exists. Unlike `process_info/[1,2]`, `is_process_alive/1` does not report zombie processes as alive.

`length(List)`

Returns the length of `List`.

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

Allowed in guard tests.

Failure: `badarg` if the argument is not a proper list.

`link(Pid)`

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

Failure: `badarg` if the argument is not a `Pid` or port. Sends the EXIT signal `noproc` to the process which evaluates `link` if the argument is the `Pid` of a process which does not exist.

`list_to_atom(CharIntegerList)`

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

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

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

`list_to_binary(List)`

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

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

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

`list_to_float(AsciiIntegerList)`

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

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

Failure: `badarg` if the argument is not a list of integers, or if `AsciiIntegerList` contains a bad representation of a float.

`list_to_integer(AsciiIntegerList)`

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

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

Failure: `badarg` if the argument is not a list of integers, or if `AsciiIntegerList` contains a bad representation of an integer.

`list_to_pid(AsciiIntegerList)`

Returns a `Pid` whose text representation is the integers (ASCII-values) in `AsciiIntegerList`. This BIF is intended for debugging, and 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 the argument is not a list of integers, or `AsciiIntegerList` contains a bad representation of a `Pid`.

`list_to_tuple(List)`

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

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

Failure: `badarg` if `List` is not a proper list.

`erlang:load_module(Module, Binary)`

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

`badfile` If the object code in `Binary` has an incorrect format.

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

`badfile` If the object code contains code for another module than `Module`

Warning:

Code handling - which includes loading, deleting, and replacing of modules - is done by the module code in normal Erlang implementations. This BIF is intended for the implementation of the module named `code` and should not be used elsewhere.

This is an optional BIF.

Failure: `badarg` if the first argument is not an atom, or the second argument is not a binary.

`erlang:loaded()`

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

`erlang:localtime()`

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

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

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

`erlang:localtime_to_universaltime(DateTime)`

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

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

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

`make_ref()`

Returns an almost unique reference.

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

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

`erlang:make_tuple(Arity, InitialValue)`

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

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

`erlang:md5(Data) -> Digest`

Types:

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

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

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

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

`erlang:md5_final(Context) -> Digest`

Types:

- `Context = Digest = binary()`

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

`erlang:md5_init() -> Context`

Types:

- `Context = binary()`

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

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

Types:

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

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

`erlang:module_loaded(Module)`

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

Warning:

This BIF is intended for the implementation of the module named `code` and should not be used anywhere else. Use `code:is_loaded/1` instead.

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

This is an optional BIF.

Failure: `badarg` if the argument is not an atom.

`erlang:monitor(Type, Item)`

The current process starts monitoring `Item`, and will be notified when `Item` dies, with a message `{'DOWN', Ref, Type, Object, Info}`, where `Ref` is the value returned by the call to `erlang:monitor/2`, and `Info` gives additional information. The message is also sent if `Item` is already dead. `Object` refers to the same entity as `Item`, but sometimes with a different format e.g. when `Item` is a process name `Object` may be the pid. The value returned can be used for disabling the monitor (see `erlang:demonitor/1`).

The currently allowed value for `Type` is the atom `process`. `Item` may then be a pid, an atom `Name` or a tuple `{Name, Node}` where `Node` also is an atom; `Info` in the message is the exit reason of the process (or `noproc` or `noconnection`, when the process does not exist or the remote node goes down, respectively, in analogy with `link/1`). `Object` in the message is the pid that `Item` refers to, but if `Name` is not registered on the referred node or if `Node` is not alive then `Object` is equal to `Name`. If an attempt is made to monitor a process on an older node (where remote process monitoring is not implemented (or one where remote process monitoring by registered name is not implemented)), the call fails with `badarg`.

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

`monitor_node(Node, Flag)`

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

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

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

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

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

Returns `true`.

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

`node()`

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

Allowed in guard tests.

`node(Arg)`

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

Allowed in guard tests.

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

`nodes()`

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

`nodes(Arg)`

Types:

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

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

ArgAtom description:

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

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

`connected` Nodes connected to this node.

`this` This node.

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

More ArgAtoms may be added in the future.

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

Failure: badarg if argument isn't an ArgAtom or a list of ArgAtoms.

`now()`

Returns the tuple `{MegaSecs, Secs, Microsecs}`

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

`open_port(PortName, PortSettings)`

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

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

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

Atom *This use of open_port() is obsolete and will be removed in a future version of Erlang. Use the file module instead.* The atom is assumed to be the name of an external resource. A transparent connection is established between Erlang and the resource named by the atom `Atom`. The characteristics of the port depend on the type of resource. If `Atom` represents a normal file, the entire contents of the file is sent to the Erlang process as one or more messages. When messages are sent to the port, it causes data to be written to the file.

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

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

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

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

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

`N` specifies the maximum line length in bytes. Lines longer than this will be delivered in more than one message, with the `Flag` set to `noeof` 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 `noeof`. In all other cases, lines are delivered with `Flag` set to `eof`.

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

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

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

`exit_status` This is only valid for `{spawn, Command}` where `Command` refers to an external program. When the external process connected to the port exits, a message of the form `{Port, {exit_status, Status}}` is sent to the connected process, where `Status` is the exit status of the external process. If the program aborts, on Unix the same convention is used as the shells do (i.e., `128+signal`). If the `eof` option has been given as well, the `eof` message and the `exit_status` message appear in an unspecified order. If the port program closes its stdout without exiting, the `exit_status` option will not work.

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

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

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

`in` The port can only be used for input.

`out` The port can only be used for output.

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

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

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

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

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

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

`enametoolong` The external command given was too long.

`emfile` There are no more available file descriptors.

`enfile` A file or port table is full.

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

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

`erlang:phash(Term, Range)`

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

This BIF should always be used instead of the old deprecated `erlang:hash/2` BIF, as it calculates better hashes for all datatypes.

`pid_to_list(Pid)`

Returns a list which corresponds to the process `Pid`.

Warning:

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

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

Failure: `badarg` if the argument is not a `Pid`.

`port_close(Port)`

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

Returns: `true`.

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

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

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

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

`port_command(Port, Data)`

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

Returns: `true`.

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

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

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

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

`port_connect(Port, Pid)`

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

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

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

Returns: `true`.

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

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

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

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

`port_control(Port, Operation, Data)`

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

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

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

`port_call(Port, Operation, Data)`

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

`Port` is an erlang port, referring to a driver.

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

`Data` is any erlang term. This data is converted to binary term format and send to the port.

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

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

`erlang:port_info(Port, Item)`

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

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

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

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

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

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

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

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

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

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

`erlang:port_to_list(Port)`

Returns a list which corresponds to the port identifier `Port`.

Warning:

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

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

Failure: `badarg` if the argument is not a port.

`erlang:ports()`

Returns a list of all ports on the current node.

`erlang:pre_loaded()`

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

`erlang:process_display(Pid, Type)`

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

`process_flag(Flag, Option)`

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

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

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

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

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

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

`process_flag(Pid, Flag, Option)`

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

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

`process_info(Pid)`

Returns a long list which contains information about the process `Pid`. This BIF is only intended for debugging. It should not be used for any other purpose. The list returned contains the following tuples. The order in which these tuples are returned is not defined, nor are all the tuples mandatory.

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

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

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

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

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

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

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

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

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

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

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

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

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

`{status, Status}` Status is the status of the process. Status is waiting (waiting for a message), running, runnable (ready to run, but another process is running), suspended (suspended on a “busy” port or by the `erlang:suspend_process/1` BIF), or exiting (if the process has exited, but remains as a zombie).

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

Failure: `badarg` if the argument is not a Pid, or if Pid is a Pid of a remote process.

`process_info(Pid, Item)`

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

Returns undefined if no information is known about the process.

`process_info` can be used to obtain information about processes which have exited but whose data are still kept, so called zombie processes. To determine whether to keep information about dead processes, use the BIF `erlang:system_flag/2`. Since `process_info` does not necessarily return undefined for a dead process, use `is_process_alive/1` to check whether a process is alive.

Item `exit` returns `[]` if the process is alive, or `{exit, Reason}` if the process has exited, where Reason is the exit reason.

Item `registered_name` returns `[]` if the process has no registered name. If the process is a zombie, the registered name it had when it died is returned.

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

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

Item `last_calls` returns `false` if call saving is not active for the process (see `process_flag/3` [page 93]). If call saving is active, a list is returned, in which the last element is the most recent.

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

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

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

Not all implementations support every one of the above Items.

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

`processes()`

Returns a list of all processes on the current node, including zombie processes. See `system_flag/2` [page 101].

```
> processes().  
[<0.0.1>, <0.1.1>, <0.2.1>, <0.3.1>, <0.4.1>, <0.6.1>]
```

`erlang:purge_module(Module)`

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

Warning:

In normal Erlang implementations, code handling - which is loading, deleting and replacing modules - is evaluated by the module code. This BIF is intended to be used by the implementation of the module code and should not be used in any other place.

This is an optional BIF.

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

`put(Key, Value)`

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

Note:

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

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

`erlang:read_timer(Ref)`

`returns_timer(Ref)` returns the number of milliseconds remaining for a timer, where `Ref` was returned by either `send_after/3` or `start_timer/3`. If the timer was active, `read_timer/1` returns the time in milliseconds left until the timer will expire, otherwise `false` (which may mean that `Ref` was never a timer, or that it has been cancelled, or that it has already delivered its message).

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

`erlang:ref_to_list(Ref)`

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

Warning:

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

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

Failure: `badarg` if the argument is not a reference.

`register(Name, P)`

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

Returns `true`.

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

`registered()`

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

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

`erlang:resume_process(Pid)`

Resume a suspended process. This should be used for debugging purposes only, not in production code.

`round(Number)`

Returns an integer by rounding the number `Number`. Allowed in guard tests.

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

Failure: `badarg` if the argument is not a float (or an integer).

`self()`

Returns the process identity of the calling process. Allowed in guard tests.

```
> self().  
<0.16.1>
```

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

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

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

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

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

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

`erlang:set_cookie(Node, Cookie)`

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

`setelement(Index, Tuple, Value)`

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

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

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

`size(Item)`

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

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

Allowed in guard tests.

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

`spawn(Fun)`

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

`spawn(Node, Fun)`

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

`spawn(Module, Function, ArgumentList)`

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

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

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

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

`spawn(Node, Module, Function, ArgumentList)`

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

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

`spawn_link(Fun)`

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

`spawn_link(Node, Fun)`

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

Returns the Pid of the newly created process.

Failure: See `spawn/3`.

`spawn_link(Module, Function, ArgumentList)`

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

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

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

Returns the Pid of the newly created process.

Failure: See `spawn/3`.

`spawn_link(Node, Module, Function, ArgumentList)`

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

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

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

Warning:

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

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

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

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

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

Here are a few cases when it could be useful to change `fullsweep_after`. Firstly, if you want binaries that are no longer used to be thrown away as soon as possible. (Set `Number` to zero.) Secondly, a process that mostly have short-lived data will be fullswept seldom or never, meaning that the old heap will contain mostly garbage. To ensure a fullsweep once in a while, set `Number` to a suitable value such as 10 or 20. Thirdly, in embedded systems with limited amount of RAM and no virtual memory, you might want to preserve memory by setting `Number` to zero. (You probably want to set the value globally. See `system_flag/2` [page 101].)

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

`split_binary(Binary, Pos)`

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

```

1> B = list_to_binary("0123456789").
#Bin
2> size(B).
10
3> {B1, B2} = split_binary(B,3).
{#Bin, #Bin}
4> size(B1).
3
5> size(B2).
7

```

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

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

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

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

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

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

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

`statistics(Type)`

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

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

`runtime` Returns `{Total_Run_Time, Time_Since_Last_Call}`.

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

`reductions` Returns `{Total_Reductions, Reductions_Since_Last_Call}`.

`garbage_collection` Returns `{Number_of_GC's, Words_Reclaimed, 0}`. This information may not be valid for all implementations.

All times are in milliseconds.

```

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

```

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

`erlang:suspend_process(Pid)`

Suspend a process. This should be used for debugging purposes only, not in production code.

`erlang:system_flag(Flag, Value)`

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

The following values for `Flag` are currently allowed: `keep_zombies`, `fullsweep_after`, and `backtrace_depth`.

The value of the `keep_zombies` flag is an integer which indicates how many processes to keep in memory when they exit, so that they can be inspected with `process_info`. Originally, the number is 0. Setting it to 0 disables the keeping of zombies. A negative number `-N` means to keep the `N` latest zombies; a positive value `N` means to keep the `N` first zombies. Setting the flag always clears away any already saved zombies. The maximum number of zombies which can be saved is 100. Resources owned by a zombie process are cleared away immediately when the process dies, for example ets tables and ports, and cannot be inspected.

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

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

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

`erlang:system_info(What)`

What can be any of the atoms `info`, `procs`, `loaded`, `dist`, `thread_pool_size`, `allocated_areas`, or `allocator`. The BIF returns information of the different ‘topics’ as binary data objects (except for `thread_pool_size`, `allocated_areas`, and `allocator` see below).

`erlang:system_info(thread_pool_size)` Returns the number of threads used for driver calls (as an integer).

`erlang:system_info(allocated_areas)` Returns a list of tuples. Each tuple contains an atom describing the type of memory as first element and the amount of allocated memory in bytes as second element. In those cases when the system pre-allocate memory, a third element is present. This third element contains the amount of used memory in bytes.

A lot of these values are shown by the `(i)nfo` alternative under the `BREAK` menu. The `BREAK` menu can be reached by typing `Control C` in the Erlang shell.

Observe that this is not a complete list of memory allocated by the system!

`erlang:system_info(allocator)` Returns `{Allocator, Version, Features, Settings}`.

Types:

- `Allocator` = `atom()`
- `Version` = `[int()]`
- `Features` = `[atom()]`

- `Settings = [{Setting, Value}]`
- `Setting = atom()`
- `Value = term()`

Explanation:

- `Allocator` corresponds to the `malloc()` implementation used. If `Allocator` equals undefined, the `malloc()` implementation used couldn't 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. Currently `sl_alloc`, `mmap`, and `fix_alloc` are identified.
- `Settings` is a list of setting/value pairs accepted by the sub-components that they have been passed to. The actual setting/value pair used may have been modified by the sub-component. Settings may differ between different combinations of platforms, allocators, and allocation features. Memory sizes are given in bytes. See also "System Flags" in `erl(1)`.

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

`term_to_binary(Term)`

This BIF returns the encoded value of any Erlang term and turns it into the Erlang external term format. It can be used for a variety of purposes, for example writing a term to a file in an efficient way, or sending an Erlang term to some type of communications channel not supported by distributed Erlang.

Returns a binary data object which corresponds to an external representation of the Erlang term `Term`.

`term_to_binary(Term, Options)`

This BIF returns the encoded value of any Erlang term and turns it into the Erlang external term format. If the `Options` list contains the atom `compressed`, the external term format will be compressed. The compressed format is automatically recognized by `binary_to_term/1` in R7.

Returns a binary data object which corresponds to an external representation of the Erlang term `Term`.

Failure: `badarg` if `Options` is not a list or if contains something else than the supported flags (currently only the atom `compressed`).

`throw(Any)`

A non-local return from a function. If evaluated within a `catch`, `catch` will return the value `Any`.

```
> catch throw({hello, there}).
{hello, there}
```

Failure: `nocatch` if not evaluated within a `catch`.

`time()`

Returns the tuple `{Hour, Minute, Second}` of the current system time. The time zone correction is implementation-dependent.

```
> time().
{9, 42, 44}
```

`tl(List)`

Returns `List` stripped of its first element.

```
> tl([geesties, guilies, beasties]).
[guilies, beasties]
```

Failure: `badarg` if `List` is the empty list `[]`, or is not a list. Allowed in guard tests.

`erlang:trace(PidSpec, How, Flaglist)`

Turns on (if `How == true`) or off (if `How == false`) the trace flags in `Flaglist` for the process or processes represented by `PidSpec`. `PidSpec` is either a pid for a local process, or one of the following atoms:

`existing` All processes currently existing.

`new` All processes that will be created in the future.

`all` All currently existing processes and all processes that will be created in the future.

`Flaglist` can contain any number of the following atoms (the “message tags” refers to the list of messages following below):

`all` All trace flags.

`send` Traces the messages the process `Pid` sends. Message tags: `send`, `send_to_non_existing_process`.

`'receive'` Traces the messages the process `Pid` receives. Message tags: `'receive'`.

`procs` Traces process related events, for example `spawn`, `link`, `exit`. Message tags: `spawn`, `exit`, `register`, `unregister`, `link`, `unlink`, `getting_linked`, `getting_unlinked`.

`call` Traces function calls to functions that tracing has been enabled for. Use the `erlang:trace_pattern/3` [page 106] BIF to enable tracing for functions. Message tags: `call`, `return_from`.

`silent` To be used in conjunction with the `call` trace flag. Sets the call trace message mode for the process `Pid` to *silent*, i.e., the call tracing is active, match specs are executed as normal, but no call trace messages are generated.

The *silent* mode can, of course, be inhibited by executing `erlang:trace/3` without the `silent` flag, but also by a match spec executing the `{silent, false}` function.

`return_to` Traces the actual return of a process from a traced function back to its caller. This return trace only works together with `call` trace and functions traced with the `local` option to `erlang:trace_pattern/3` [page 106]. The semantics is that a message is sent when a call traced function actually returns, i.e., when a chain of tail recursive calls is ended. There will be only one trace message sent per chain of tail recursive calls, why the properties of tail recursiveness for function calls are kept while tracing with this flag. Using `call` and `return_to` trace together makes it possible to know exactly in which function a process executes at any time.

To get trace messages containing return values from functions, use the `{return_trace}` match_spec action instead.

Message tags: `return_to`.

running Traces scheduling of processes. Message tags: in, out.

garbage_collection Traces garbage collections of processes. Message tags: gc_start, gc_end.

timestamp Make a time stamp in all trace messages. The time stamp (Ts) is of the same form as returned by `erlang:now()`.

arity Instead of {Mod, Fun, Args} in call traces, there will be {Mod, Fun, Arity}.

set_on_spawn Makes any process created by Pid inherit the flags of Pid, including the set_on_spawn flag.

set_on_first_spawn Makes the first process created by Pid inherit the flags of Pid. That process does not inherit the set_on_first_spawn flag.

set_on_link Makes any process linked by Pid inherit the flags of Pid, including the set_on_link flag.

set_on_first_link Makes the first process linked to by Pid inherit the flags of Pid. That process does not inherit the set_on_first_link flag.

{tracer, Tracer} Tracer should be the pid for a local process or the port identifier for a local port. All trace messages will be sent to the given process or port. If this flag is not given, trace messages will be sent to the process that called `erlang:trace/3`.

The effect of combining `set_on_first_link` with `set_on_link` is the same as having `set_on_first_link` alone. Likewise for `set_on_spawn` and `set_on_first_spawn`.

If the `timestamp` flag is not given, the tracing process will receive the trace messages described below. If the `timestamp` flag is given, the first element of the tuple will be `trace_ts` and the timestamp will be in the last element of the tuple.

{trace, Pid, 'receive', Message} When the traced Pid receives something.

{trace, Pid, send, Msg, To} When Pid sends a message.

{trace, Pid, send_to_non_existing_process, Msg, To} When Pid sends a message to a non existing process.

{trace, Pid, call, {M,F,A}} When Pid makes a function/BIF call. The return values of calls are never supplied, only the call and its arguments.

{trace, Pid, return_to, {M,F,A}} When Pid returns *to* function {M,F,A}. This message will be sent if both the `call` and the `return_to` flags are present and the function is set to be traced on *local* function calls. The message is only sent when returning from a chain of tail recursive function calls where at least one call generated a call trace message (i.e., the functions match specification matched and {message,false} was not an action).

{trace, Pid, return_from, {M,F,A}, ReturnValue} When Pid returns *from* the function {M,F,A} This trace message is sent when the `call` flag has been specified, and the function has a match specification with a `return_trace` action.

{trace, Pid, spawn, Pid2, {M, F, A}} When Pid spawns a new process Pid2. {M, F, A} are the initial function call with arguments for the new process.

Note that A is supposed to be the argument list, but may be any term in the case of an erroneous spawn.

{trace, Pid, exit, Reason} When Pid exits with reason Reason.

{trace, Pid, link, Pid2} When Pid links to a process Pid2.

{trace, Pid, unlink, Pid2} When Pid removes the link from a process Pid2.

`{trace, Pid, getting_linked, Pid2}` When `Pid` gets linked to a process `Pid2`.

`{trace, Pid, getting_unlinked, Pid2}` When `Pid` gets unlinked from a process `Pid2`.

`{trace, Pid, register, Name}` When `Pid` gets the name `Name` registered.

`{trace, Pid, unregister, Name}` When `Pid` gets the name `Name` unregistered. Note that this is done automatically when a registered process exits.

`{trace, Pid, in, {M,F,A} | 0}` When `Pid` is scheduled to run. The process will run in function `{M,F,A}`, where `A` is always the arity. On some rare occasions the current function cannot be determined, then the last element is 0.

`{trace, Pid, out, {M,F,A} | 0}` When `Pid` is scheduled out. The process was running in function `{M,F,A}` where `A` is always the arity. On some rare occasions the current function cannot be determined, then the last element is 0.

`{trace, Pid, gc_start, Info}` Sent when garbage collection is about to be started. `Info` is a list of two-element tuples, where the first element is a key, and the second is the value. You should not depend on the tuples have any defined order. Currently, the following keys are defined.

- `heap_size` The size of the used part of the heap.
- `old_heap_size` The size of the used part of the old heap.
- `stack_size` The actual size of the stack.
- `recent_size` The size of the data that survived the previous garbage collection.
- `mbuf_size` The combined size of message buffers associated with the process.

All sizes are in words.

`{trace, Pid, gc_end, Info}` Sent when garbage collection is finished. `Info` contains the same kind of list as in the `gc_start` message, but the sizes reflect the new sizes after garbage collection.

If the tracing process dies, the flags will be silently removed.

Only one process can trace a particular process. For this reason, attempts to trace an already traced process will fail.

Returns: A number indicating the number of processes that matched `PidSpec`. If `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: `badarg` if bad arguments are given.

`erlang:trace_info(PidOrFunc, Item)`

Returns trace information about a process or exported 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 an exported function, `PidOrFunc` should be a three-element tuple: `{Module, Function, Arity}` or the atom `on_load`. No wildcards are allowed. `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 this function is not traced at all.

`match_spec` Return the match specification for this function, if it has one. If the function is not traced, the returned value is `false` and if the function is traced but has no match specification defined, the returned value is `[]`.

The actual return value will be `{Item, Value}`, where `Value` is the requested information as described above. If a pid for a dead process was given, or the name of a non-existing function, `Value` will be `undefined`.

If `PidOrFunc` is the `on_load`, the information returned refers to the default value for code that will be loaded.

`erlang:trace_pattern(MFA, MatchSpec)`

The same as `erlang:trace_pattern(MFA, MatchSpec, [])`, retained for backward compatibility.

`erlang:trace_pattern(MFA, MatchSpec, FlagList)`

This BIF is used to enable or disable call tracing for exported functions. It must be combined with `erlang:trace/3` [page 103] 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 103] to add or remove one or more processes to the set of traced processes. Use `erlang:trace_pattern/2` to add or remove exported functions to the set of traced functions.

The `erlang:trace_pattern/3` BIF can also add match specifications to an exported function. A match specification comprises a pattern that the arguments to the function must match, a guard expression which must evaluate to `true` and action to be performed. The default action is to send a trace message. If the pattern does not match or the guard fails, the action will not be executed.

The `MFA` argument should be a tuple like `{Module, Function, Arity}` or the atom `on_load` (described below). It can be the module, function, and arity for an exported function (or a BIF in any module). The `'_'` atom can be used to mean any of that kind. Wildcards can be used in any of the following ways:

`{Mod, Func, '_'}` All exported functions of any arity named `Func` in module `Mod`.

`{Mod, '_', '_'}` All exported functions in module `Mod`.

`{'_', '_', '_'}` All exported functions in all loaded modules.

Other combinations, such as `{Mod, '_', Arity}`, are not allowed. Local functions will match wildcards only if the `local` option is in the `FlagList`.

If the MFA argument is the atom `on_load`, the match specification and flag list will be used on all modules that are newly loaded.

The `MatchSpec` argument can take any of the following forms:

`false` Disable tracing for the matching function(s). Any match specification will be removed.

`true` Enable tracing for the matching function(s).

`MatchSpecList` A list of match specifications. An empty list is equivalent to `true`. See the ERTS User's Guide for a description of match specifications.

The `FlagList` parameter is a list of options. The following options are allowed:

`global` Turn on or off call tracing for global function calls (i.e., calls specifying the module explicitly). Only exported functions will match and only global calls will generate trace messages. This is the default.

`local` Turn on or off call tracing for all types of function calls. Trace messages will be sent whenever any of the specified functions are called, regardless of how it is 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.

The options are mutually exclusive and `global` is the default (if no options are specified). A function can be *either* globally or locally traced. If global trace is specified for a specified set of functions, local trace 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).

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 105] BIF to retrieve the existing match specification.

Returns the number of exported functions that matched the MFA argument. This will be zero if none matched at all.

Failure: `badarg` for invalid MFA or `MatchSpec`.

`trunc(Number)`

Returns an integer by the truncation of `Number`. Allowed in guard tests.

```
> trunc(5.5).
5
```

Failure: `badarg` if the argument is not a float, or an integer.

`tuple_to_list(Tuple)`

Returns a list which corresponds to `Tuple`. `Tuple` may contain any valid Erlang terms.

```
> tuple_to_list({share, {'Ericsson.B', 163}}).
[share, {'Ericsson.B', 163}]
```

Failure: badarg if the argument is not a tuple.

`erlang:universaltime()`

Returns the current date and time according to Universal Time Coordinated (UTC), also called GMT, in the form `{{Year, Month, Day}, {Hour, Minute, Second}}` if supported by the underlying operating system. If not, `erlang:universaltime()` is equivalent to `erlang:localtime()`.

```
> erlang:universaltime().  
{{1996,11,6},{14,18,43}}
```

`erlang:universaltime_to_localtime(DateTime)`

Converts UTC date and time in `DateTime` to local date and time if supported by the underlying operating system. Otherwise, no conversion is done, and `DateTime` is returned. The return value is of the form `{{Year, Month, Day}, {Hour, Minute, Second}}`.

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

```
> erlang:universaltime_to_localtime({{1996,11,6},{14,18,43}}).  
{{1996,11,7},{15,18,43}}
```

`unlink(Pid)`

Removes a link, if there is one, from the calling process to another process given by the argument `Pid`.

Returns `true`. Will not fail if not linked to `Pid`, or if `Pid` does not exist.

Failure: badarg if the argument is not a valid `Pid`.

`unregister(Name)`

Removes the registered name for a port or process, given by the atom argument `Name`.

Returns the atom `true`.

```
> unregister(db).  
true
```

Failure: badarg if `Name` is not the name of a registered port or process.

Users are advised not to unregister system processes.

`whereis(Name)`

Returns the identifier of the port or process registered under `Name` (see `register/2`).

Returns `undefined` if no such port or process is registered.

```
> whereis(user).  
<0.3.1>
```

Failure: badarg if the argument is not an atom.

`erlang:yield()`

Voluntarily let other processes (if any) get a chance to execute. Using `yield()` is similar to receive after `1 -> ok end`, except that `yield()` is faster.

error_handler

Erlang Module

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

Exports

`undefined_function(Module, Func, ArgList) -> term()`

Types:

- `Module = Func = atom()`
- `ArgList = [term()]`

This function is evaluated if a call is made to `Module:Func(ArgList)` which is undefined. This function is evaluated inside the process making the original call.

If `Module` is interpreted, the interpreter is invoked and the return value of the interpreted `Func(ArgList)` call is returned.

Otherwise, it returns, if possible, the value of `apply(Module, Func, ArgList)` after an attempt has been made to autoload `Module`. If this is not possible, the function calling `Module:Func(ArgList)` is exited.

`undefined_lambda(Module, Fun, ArgList) -> term()`

Types:

- `Module = Fun = atom()`
- `ArgList = [term()]`

This function is evaluated if a call is made to `Fun(ArgList)` when the module defining the `fun` is not loaded. This function is evaluated inside the process making the original call.

If `Module` is interpreted, the interpreter is invoked and the return value of the interpreted `Fun(ArgList)` call is returned.

Otherwise, it returns, if possible, the value of `apply(Fun, ArgList)` after an attempt has been made to autoload `Module`. If this is not possible, the process calling the `fun` is exited.

Notes

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

Changes in the code which may seem small can cause a deadlock as unforeseen consequences may occur. The use of `input` is dangerous in this type of code.

error_logger

Erlang Module

The error logger is an event manager behaviour which runs with the registered name `error_logger` (see more about event managers/handlers in the Design Principles chapter and in `gen_event(3)`). All error messages from the Erlang runtime system are sent to this process as messages with the format `{emulator, Gleader, Str}`, where `Str` is a string which describes the error in plain English. The `Gleader` argument is the group leader process of the process causing the error. This is useful in a distributed setting as all error messages can be returned to the `error_logger` process on the originating node.

All errors detected by the standard libraries are reported with the `error_logger` functions. Errors detected in application modules should also be reported through the `error_logger` in order to get uniform reports.

Associated event handlers can be used to add private types of reports to the `error_logger`. An event handler which recognizes the specialized report type is first added to the `error_logger` (`add_report_handler/1,2`)

The standard configuration of the `error_logger` supports the logging of errors to the `tty`, or to a specified file. There is also a multi-file logger which logs all events, not only the standard error events, to several files. (see `log_mf.h(3)`).

All error events are tagged with the group leader `Gleader` in order to send the error to the originating node.

Exports

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

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

Types:

- `Pid` = `pid()`
- `What` = `{already_started, Pid}` | `term()`

Starts the `error_logger`. The `start_link` function should be used when the `error_logger` is supervised

```
error_report(Report) -> ok
```

Types:

- `Report` = `[{Tag, Data}]` | `[term()]` | `string()` | `term()`
- `Tag` = `term()`
- `Data` = `term()`

Sends a standard error report event to the error logger. This report event is handled by the standard event handler. The report is formatted as follows:

```
Tag1:   Data1
Tag2:   Data2
Term1
Term2
```

If Report is a string(), the string is written.

The report is written with an error heading.

```
error_report(Type,Report) -> ok
```

Types:

- Type = term()
- Report = [{Tag, Data}] | [term()] | string() | term()
- Tag = term()
- Data = term()

Sends a user defined error report type event to the error logger. If specialized error handling is required, an event handler recognizing this Type of report must first be added to the error_logger.

It is recommended that the Report follows the same structure as error_report/1 above.

```
info_report(Report) -> ok
```

Types:

- Report = [{Tag, Data}] | [term()] | string() | term()
- Tag = term()
- Data = term()

Sends an information report to the error logger. This report event is handled by the standard event handler. The report is formatted as follows:

```
Tag1:   Data1
Tag2:   Data2
Term1
Term2
```

If Report is a string(), the string is written.

The report is written with an information heading.

```
info_report(Type,Report) -> ok
```

Types:

- Type = term()
- Report = [{Tag, Data}] | [term()] | string() | term()
- Tag = term()
- Data = term()

Sends a user defined information report type event to the error logger. If specialized error handling is required, an event handler recognizing this Type of report must first be added to the `error_logger`.

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

```
error_msg(Format) -> ok
error_msg(Format,Args) -> ok
format(Format,Args) -> ok
```

Types:

- Format = string()
- Args = [term()]

Sends an error event to the error logger. The Format and Args arguments are the same as the arguments of `io:format/2`. These events are handled by the standard event handler.

```
info_msg(Format) -> ok
info_msg(Format,Args) -> ok
```

Types:

- Format = string()
- Args = [term()]

Sends an information event to the error logger. The Format and Args arguments are the same as the arguments of `io:format/2`. These events are handled by the standard event handler.

```
tty(Flag) -> ok
```

Types:

- Flag = true | false

Enables or disables error printouts to the tty. If Flag is false, all text that the error logger would have sent to the terminal is discarded. If Flag is true, error messages are sent to the terminal screen.

```
logfile(Request) -> ok | FileName | {error, What}
```

Types:

- Request = {open, FileName} | close | filename
- FileName = atom() | string()
- What = term()

This function makes it possible to append a copy of all standard error printouts to a file. It can be used in combination with the `tty(false)` function in to have a silent system, where all errors are logged to a file.

Request can be:

- {open, Filename}. Opens the file Filename to store a copy of all error messages. Returns ok, or {error, What}.
- close. Closes the current log file. Returns ok, or {error, What}.

- `filename`. Returns `{error, What}` or `FileName`, where `FileName` is the name of the open log file.

There can only be one active log file.

```
add_report_handler(Module) -> ok | Other
```

```
add_report_handler(Module,Args) -> ok | Other
```

Types:

- `Module` = `atom()`
- `Args` = `term()`
- `Other` = `term()`

Adds a new event handler to the error logger. The event handler is initialized by a call to the `Module:init/1` function. This function must return `{ok, State}`. If anything else (`Other`) is returned, the handler is not added.

The report (event) handler will be called for every error event that the error logger receives (`Module:handle_event/2`). Errors dedicated to this handler should be handled accordingly.

```
delete_report_handler(Module) -> Return | {error, What}
```

Types:

- `Module` = `atom()`
- `Return` = `term()`
- `What` = `term()`

Deletes an error report (event) handler. The `Module:terminate/2` function is called in order to finalize the event handler. The return value of the `terminate/2` function is `Return`.

```
swap_handler(ToHandler) -> ok
```

Types:

- `ToHandler` = `tty` | `{logfile, File}`
- `File` = `atom()` | `string()`

The `error_logger` event manager is initially started with a primitive event handler which buffers and prints the raw error events. However, this function does install the standard event handler to be used according to the system configuration.

Events

The error logger event manager forwards the following events to all added event handlers. In the events that follow, Gleader is the group leader process identity of the error reporting process, and EPid is the process identity of the `error_logger`. All other variables are described with the function in which they appear.

`{error_report, Gleader, {EPid, std_error, Report}}` This event is generated when the `error_report/1` function is called.

`{error_report, Gleader, {EPid, Type, Report}}` This event is generated when the `error_report/2` function is called.

`{info_report, Gleader, {EPid, std_info, Report}}` This event is generated when the `info_report/1` function is called.

`{info_report, Gleader, {EPid, Type, Report}}` This event is generated when the `info_report/2` function is called.

`{error, Gleader, {EPid, Format, Args}}` This event is generated when the `error_msg` or `format` functions are called.

`{info_msg, Gleader, {EPid, Format, Args}}` This event is generated when the `info_msg` functions are called.

`{info, Gleader, {EPid, term(), []}}` This structure is only used by the `init` process for erroneously received messages.

`{emulator, Gleader, string()}` This event is generated by the runtime system. If the error was not issued by a special process, Gleader is `noproc`. This event should be handled in the `handle_info/2` function of the event handler.

Note:

All events issued by a process which has the group leader Gleader process located on another node will be passed to this node by the `error_logger`.

See Also

`gen_event(3)`, `log_mf_h(3)`

file

Erlang Module

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

Most functions have a name argument such as a file name or directory name, which is either an atom, a string, or a deep list of characters and atoms. A path is a list of directory names. If the functions are successful, they return `ok`, or `{ok, Value}`.

If an error occurs, the return value has the format `{error, Reason}`. Reason is an atom which is named from the Posix error codes used in Unix, and in the runtime libraries of most C compilers. In the following descriptions of functions, the most typical error codes are listed. By matching the error code, applications can use this information for error recovery. To produce a readable error string, use `format_error/1`.

On operating systems with thread support (Solaris and Windows), it is possible to let file operations be performed in threads of their own, allowing other Erlang processes to continue executing in parallel with the file operations. See the command-line option `+A` in the manual page for `erl`.

Exports

`change_group(Filename, Gid)`

Change group of a file. See `write_file_info/2`.

`change_owner(Filename, Uid)`

Change owner of a file. See `write_file_info/2`.

`change_owner(Filename, Uid, Gid)`

Change owner and group of a file. See `write_file_info/2`.

`change_time(Filename, Mtime)`

Change the modification and access times of a file. See `write_file_info/2`.

`change_time(Filename, Mtime, Atime)`

Change the modification and access times of a file. See `write_file_info/2`.

`close(IODevice)`

Closes the file referenced by `IoDevice`. It mostly returns `ok`, expect for some severe errors such as out of memory.

Note that if the option `delayed_write` was used when opening the file, `close/1` might return an old write error and not even try to close the file. See `open/2`.

`consult(Filename)`

Opens file `Filename` and reads all the Erlang terms in it. Returns one of the following:

- `{ok, TermList}` The file was successfully read.
- `{error, Atom}` An error occurred when opening the file or reading it. The `Atom` is a Posix error code. See the description of `open/2` for a list of typical error codes.
- `{error, {Line, Mod, Term}}` An error occurred when interpreting the Erlang terms in the file. Use the `format_error/1` function to convert the three-element tuple to an English description of the error.

`copy(Source, Destination)`

Copies the contents of `Source` to `Destination`. `Source` and `Destination` are either filenames or open file references from e.g `open/2`.

The same as `copy/3` but with infinite byte count.

`copy(Source, Destination, ByteCount)`

Copies `ByteCount` bytes from `Source` to `Destination`. `Source` and `Destination` are either filenames or open file references from e.g `open/2`.

If `Source` is a tuple `{Filename, ModeList}` where `ModeList` is a mode list as for `open/2`, the source is opened with `read` mode prepended to the mode list before the copy, and closed when done.

If `Source` is a filename, it is interpreted as `{Source, []}`. The file is thereby read from the beginning.

If `Destination` is a tuple `{Filename, ModeList}` where `ModeList` is a mode list as for `open/2`, the destination is opened with `write` mode prepended to the mode list before the copy, and closed when done.

If `Destination` is a filename, it is interpreted as `{Destination, []}`. This implies that the previous file contents are overwritten.

If both `Source` and `Destination` are filenames or `{Filename, ModeList}` tuples, the files are opened with `[raw, read, binary]` and `[raw, write, binary]` prepended to their mode lists, respectively, to optimize the copy.

Returns `{ok, BytesCopied}` where `BytesCopied` is the number of bytes that actually was copied, which may be less than `ByteCount` if end of file was encountered on the source. If the operation fails, `{error, Reason}` is returned.

Typical error reasons: As for `open/2` if a file had to be opened, and as for `read/2` and `write/2`.

`del_dir(DirName)`

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

Typical error reasons are:

`eaccess` Missing search or write permissions for the parent directories of `DirName`.
`eeexist` The directory is not empty.
`enoent` The directory does not exist.
`enotdir` A component of `DirName` is not a directory. On some platforms, `enoent` is returned instead.
`EINVAL` Attempt to delete the current directory. On some platforms, `eaccess` is returned instead.

`delete(Filename)`

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

Typical error reasons are:

`ENOENT` The file does not exist.
`EACCESS` Missing permission for the file or one of its parents.
`EPERM` The file is a directory and the user is not super-user.
`ENOTDIR` A component of the file name is not a directory. On some platforms, `ENOENT` is returned instead.

`eval(Filename)`

Opens the file `Filename` and evaluates all the expression sequences in it. It returns one of the following:

`ok` The file was read and evaluated. The actual result of the evaluation is not returned; any expression sequence in the file must be there for its side effect.
`{error, Atom}` An error occurred when opening the file or reading it. The `Atom` is a Posix error code. See the description of `open/2` for a list of typical error codes.
`{error, {Line, Mod, Term}}` An error occurred when interpreting the Erlang terms in the file. Use the `format_error/1` function to convert the three-element tuple to an English description of the error.

`file_info(Filename)`

Note:

This function is obsolete. Use `read_file_info` instead.

Retrieves information about a file. Returns `{ok, FileInfo}` if successful, otherwise `{error, Reason}`. `FileInfo` is a tuple with the following fields:

`{Size, Type, Access, AccessTime, ModifyTime, Unused1, Unused2}`

`Size` The size of the file in bytes.

`Type` The type of file which is device, directory, regular, or other.

`Access` The current system access to the file, which is one of the atoms `read`, `write`, `read_write`, or `none`.

AccessTime The last time the file was read, shown in the format {Year, Month, Day, Hour, Minute, Second}.

ModifyTime The last time the file was written, shown in the format {Year, Month, Day, Hour, Minute, Second}.

Unused1, Unused2 These fields are not used, but reserved for future expansion. They probably contain unused.

Typical error reasons: Same as for `read_file_info/1`.

`format_error(ErrorDescriptor)`

Given the error reason returned by any function in this module, it returns a descriptive string of the error in English.

`get_cwd()`

Returns {ok, CurDir}, where CurDir (a string) is the current working directory of the file server.

Note:

In rare circumstances, this function can fail on Unix. It may happen if read permission does not exist for the parent directories of the current directory.

Typical error reasons are:

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

`get_cwd(Drive)`

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

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

Typical error reasons are:

enotsup The operating system have no concept of drives.

eaccess The drive does not exist.

EINVAL The format of Drive is invalid.

`ipread_s32bu_p32bu(IoDevice, Location, MaxSize)`

Specialised indirect read function for Dets. Equivalent to `pread/3` of a header from `Location` followed by another `pread/3` of the buffer specified by the header.

Warning:

This function is not intended to be used by others than Dets. It is therefore not well documented.

`list_dir(DirName)`

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

Typical error reasons are:

`eaccess` Missing search or write permissions for `DirName` or one of its parent directories.
`enoent` The directory does not exist.

`make_dir(DirName)`

Tries to create the directory `DirName`. Missing parent directories are NOT created. Returns `ok` if successful.

Typical error reasons are:

`eaccess` Missing search or write permissions for the parent directories of `DirName`.
`eexist` There is already a file or directory named `DirName`.
`enoent` A component of `DirName` does not exist.
`enospc` There is a no space left on the device.
`enotdir` A component of `DirName` is not a directory. On some platforms, `enoent` is returned instead.

`make_link(Existing, New)`

Makes a hard link from `Existing` to `New`, on platforms that support links (Unix). This function returns `ok` if the link was successfully created, or `{error, Reason}`. On platforms that do not support links, `{error, enotsup}` will be returned.

Typical error reasons:

`eaccess` Missing read or write permissions for the parent directories of `Existing` or `New`.
`eexist` `new` already exists.
`enotsup` Hard links are not supported on this platform.

`make_symlink(Name1, Name2)`

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

Typical error reasons:

`eaccess` Missing read or write permissions for the parent directories of `Existing` or `New`.

`eexist` `new` already exists.

`enotsup` Symbolic links are not supported on this platform.

`open(Filename, ModeList)`

Opens the file `Filename` in the mode determined by `ModeList`. `ModeList` may contain one or more of the following items:

`read` The file, which must exist, is opened for reading.

`write` The file is opened for writing. It is created if it does not exist. Otherwise, it is truncated (unless combined with `read`).

`append` The file will be opened for writing, and it will be created 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 too many other file operations, for this to happen.

When this option is used, the result of `write/2` calls may prematurely be reported as successful, and if a write error should actually occur the error is reported as the result of the next file operation, which is not executed.

E.g. when `delayed_write` is used, after a number of `write/2` calls, `close/1` might return `{error, enospc}` because there was not enough space on the disc for previously written data, and `close/1` should probably be called again since the file is still open.

`delayed_write` The same as `{delayed_write, Size, Delay}` with reasonable default values for `Size` and `Delay`. (Roughly some 64 KBytes, 2 seconds)

`{read_ahead, Size}` This option activates read data buffering. If `read/2` calls are for significantly less than `Size` bytes, read operations towards the operating system are still performed for blocks of `Size` bytes. The extra data is buffered and returned in subsequent `read/2` calls, giving a performance gain since the number of operating system calls is reduced.

If `read/2` calls are for sizes not significantly less than, or even greater than `Size` bytes, no performance gain can be expected.

`read_ahead` The same as `{read_ahead, Size}` with a reasonable default value for `Size`. (Roughly some 64 KBytes)

If both `read` and `write` are specified, the file is created if it does not exist. It is not truncated if it exists.

Returns:

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

`{error, Reason}` The file could not be opened.

A file descriptor is the `Pid` of the process which handles the file. The file process is linked to the process which originally opened the file. If any process to which the file process is linked terminates, the file will be closed by the file process and the process itself will be terminated. The file descriptor returned from this call can be used as an argument to the I/O functions (see `io`).

Note:

In previous versions of `file`, modes were given as 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.

`eaccess` Missing permission for reading the file or 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.

`enospc` There is no space left on the device (if `write` access was specified).

`path_consult(Path, Filename)`

Searches the path `Path` (a list of directory names) until the file `Filename` is found. If `Filename` is an absolute file name, `Path` is ignored. The file is opened and all the terms in it are read. The function returns one of the following:

`{ok, TermList, FullName}` The file was successfully read. `FullName` is the full name of the file which was opened and read.

`{error, enoent}` The file could not be found in any of the directories in `Path`.

- `{error, Atom}` An error occurred when opening the file or reading it. The `Atom` is a Posix error code. See the description of `open/2` for a list of typical error codes.
- `{error, {Line, Mod, Term}}` An error occurred when interpreting the Erlang terms in the file. Use the `format_error/1` function to convert the three-element tuple to an English description of the error.

`path_eval(Path, Filename)`

Searches the path `Path` (a list of directory names) until the file `Filename` is found. If `Filename` is an absolute file name, `Path` is ignored. The file is opened and all the expression sequences in it are evaluated. The function returns one of the following:

- `{ok, FullName}` The file was read. `FullName` is the full name of the file which was opened and evaluated.
- `{error, enoent}` The file could not be found in any of the directories in `Path`.
- `{error, Atom}` An error occurred when opening the file or reading it. The `Atom` is a Posix error code. See the description of `open/2` for a list of typical error codes.
- `{error, {Line, Mod, Term}}` An error occurred when interpreting the Erlang terms in the file. Use the `format_error/1` function to convert the three-element tuple to an English description of the error.

`path_open(Path, Filename, Mode)`

Searches the path `Path` (a list of directory names) until the file `Filename` is found. If `Filename` is an absolute file name, `Path` is ignored. The function returns one of the following:

- `{ok, IoDevice, FullName}` The file was opened in the requested mode. `IoDevice` is a reference to the file and `FullName` is the full name of the file which was opened.
- `{error, enoent}` `Filename` was not found in the path.
- `{error, Reason}` There was an error opening `Filename`.

`position(IoDevice, Location)`

Sets the position of the file referenced by `IoDevice` to `Location`. Returns `{ok, NewPosition}` (as absolute offset) if successful, otherwise `{error, Reason}`. `Location` is one of the following:

- `{bof, Offset}` Absolute offset
- `{cur, Offset}` Offset from the current position
- `{eof, Offset}` Offset from the end of file
- `Integer` The same as `{bof, Integer}`
- `bof || cur || eof` The same as above with `Offset 0`.

Typical error reasons are:

- `EINVAL` Either the `Location` was illegal, or it evaluated to a negative offset in the file. Note that if the resulting position is a negative value you will get an error but after the call it is undefined where the file position will be.

`pread(IoDevice, [{Location, Number}, ...])`

Performs a sequence of `pread/3` in one operation, which is more efficient than calling them one at a time. Returns `{ok, [Data, ...]}` or `{error, Reason}`, where `Data` is either a list or a binary depending on the mode of the file, or `eof` if the requested position was beyond end of file.

`pread(IoDevice, Location, Number)`

Combines `position/2` and `read/2` in one operation, which is more efficient than calling them one at a time. If `IoDevice` has been opened in raw mode, some restrictions apply: `Location` is only allowed to be an integer; and the current position of the file is undefined after the operation.

`pwrite(IoDevice, [{Location, Bytes}, ...])`

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

`pwrite(IoDevice, Location, Bytes)`

Combines `position/2` and `write/2` in one operation, which is more efficient than calling them one at a time. If `IoDevice` has been opened in raw mode, some restrictions apply: `Location` is only allowed to be an integer; and the current position of the file is undefined after the operation.

`read(IoDevice, Number)`

Reads `Number` bytes from the file described by `IoDevice`. This function is the only way to read from a file opened in raw mode (although it works for normally opened files, too). Returns:

`{ok, ListOrBinary}` If the file was opened in binary mode, the read bytes are returned in a binary, otherwise in a list. The list or binary will be shorter than the the number of bytes requested if the end of the file is reached.

`eof` `eof` is returned if the `Number` was greater than zero and end of file was reached before anything at all could be read.

`{error, Reason}` A Posix error code will be returned if an error occurred.

Typical error reasons:

`ebadf` The file is not opened for reading.

`read_file(Filename)`

Returns `{ok, Binary}`, where `Binary` is a binary data object that contains the contents of `Filename`, or `{error, Reason}` if an error occurs.

Typical error reasons:

`enoent` The file does not exist.

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

`eisdir` The named file is a directory.

enotdir A component of the file name is not a directory. On some platforms, **ENOENT** is returned instead.

ENOMEM There is not enough memory for the contents of the file.

`read_file_info(Filename)`

Retrieves information about a file. Returns `{ok, FileInfo}` if successful, otherwise `{error, Reason}`. `FileInfo` is a record. Its definition can be found by including `file.hrl` from the kernel application:

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

The record contains the following fields.

size Size of file in bytes.

type The type of the file which can be device, directory, regular, or other.

access The current system access to the file, which is one of the atoms `read`, `write`, `read_write`, or `none`.

atime The last (local) time the file was read, in the format `{{Year, Month, Day}, {Hour, Minute, Second}}`.

mtime The last (local) time the file was written, in the format `{{Year, Month, Day}, {Hour, Minute, Second}}`.

ctime The interpretation of this time field depends on the operating system. On Unix, it is the last time the file or the inode was changed. In Windows, it is the create time. The format is `{{Year, Month, Day}, {Hour, Minute, Second}}`.

mode An integer which gives the file permissions as a sum of the following bit values:

```
8#00400 read permission: owner
8#00200 write permission: owner
8#00100 execute permission: owner
8#00040 read permission: group
8#00020 write permission: group
8#00010 execute permission: group
8#00004 read permission: other
8#00002 write permission: other
8#00001 execute permission: other
16#800 set user id on execution
16#400 set group id on execution
```

On Unix platforms, other bits than those listed above may be set.

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

major_device An integer which identifies the file system where the file is located. In Windows, the number indicates a drive as follows: 0 means A:, 1 means B:, and so on.

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

inode An integer which gives the inode number. On non-Unix file systems, this field will be zero.

uid An integer which indicates the owner of the file. Will be zero for non-Unix file systems.

gid An integer which gives the group that the owner of the file belongs to. Will be zero for non-Unix file systems.

Typical error reasons:

eaccess Missing search permission for one of the parent directories of the file.

enoent The file does not exist.

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

`read_link(Linkname)`

This function returns `{ok, Filename}` if `Linkname` refers to a symbolic link or `{error, Reason}` otherwise. On platforms that do not support symbolic links, the return value will be `{error, enotsup}`.

Typical error reasons:

EINVAL `Linkname` does not refer to a symbolic link.

ENOENT The file does not exist.

ENOTSUP Symbolic links are not supported on this platform.

`read_link_info(Filename)`

This function works like `read_file_info/1`, except that if `Filename` is a symbolic link, information about the link will be returned in the `file_info` record and the `type` field of the record will be set to `symlink`. If `Filename` is not a symbolic link, this function returns exactly the same result as `read_file_info/1`. On platforms that do not support symbolic link, this function is always equivalent to `read_file_info/1`.

`rename(Source, Destination)`

Tries to rename the file `Source` to `Destination`. It can be used to move files (and directories) between directories, but it is not sufficient to specify the destination only. The destination file name must also be specified. For example, if `bar` is a normal file and `foo` and `baz` are directories, `rename("foo/bar", "baz")` returns an error, but `rename("foo/bar", "baz/bar")` succeeds. Returns `ok` if it is successful.

Note:

Renaming of open files is not allowed on most platforms (see `eaccess` below).

Typical error reasons:

EACCESS 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.

`set_cwd(DirName)`

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

Typical error reasons are:

`ENOENT` The directory does not exist.

`ENOTDIR` A component of `DirName` is not a directory. On some platforms, `ENOENT` is returned.

`EACCES` Missing permission for the directory or one of its parents.

`sync(IoDevice)`

Makes sure that any buffers kept by the operating system (not by the Erlang runtime system) are written to disk. On some platforms, this function might have no effect.

Typical error reasons are:

`ENOSPC` Not enough space left to write the file.

`truncate(IoDevice)`

Truncates the file referenced by `IoDevice` at the current position. Returns `ok` if successful, otherwise `{error, Reason}`.

`write(IoDevice, Bytes)`

Writes `Bytes` (possibly a deep list of characters, or a binary) to the file described by `IoDevice`. This function is the only way to write to a file opened in raw mode (although it works for normally opened files, too).

This function returns `ok` if successful, and `{error, Reason}` otherwise.

Typical error reasons are:

`EBADF` The file is not opened for writing.

`ENOSPC` There is no space left on the device.

`write_file(Filename, Binary)`

Writes the contents of the binary data object `Binary` to the file `Filename`. The file is created if it does not exist already. If it exists, the previous contents are overwritten. Returns `ok`, or `{error, Reason}`.

Typical error reasons are:

`ENOENT` A component of the file name does not exist.

- enotdir** A component of the file name is not a directory. On some platforms, **enoent** is returned instead.
- enospc** There is a no space left on the device.
- eaccess** Missing permission for writing the file or searching one of the parent directories.
- eisdir** The named file is a directory.

`write_file_info(Filename, FileInfo)`

Change file information. Returns `ok` if successful, otherwise `{error, Reason}`. `FileInfo` is a record. Its definition can be found by including `file.hrl` from the kernel application:

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

The following fields are used from the record if they are given.

- atime** The last (local) time the file was read, in the format `{{Year, Month, Day}, {Hour, Minute, Second}}`.
- mtime** The last (local) time the file was written, in the format `{{Year, Month, Day}, {Hour, Minute, Second}}`.
- ctime** On Unix, any value give for this field will be ignored (the “ctime” for the file will be set to the current time). On Windows, this field is the new creation time to set for the file. The format is `{{Year, Month, Day}, {Hour, Minute, Second}}`.
- mode** An integer which gives the file permissions as a sum of the following bit values:
 - 8#00400** read permission: owner
 - 8#00200** write permission: owner
 - 8#00100** execute permission: owner
 - 8#00040** read permission: group
 - 8#00020** write permission: group
 - 8#00010** execute permission: group
 - 8#00004** read permission: other
 - 8#00002** write permission: other
 - 8#00001** execute permission: other
 - 16#800** set user id on execution
 - 16#400** set group id on execution

On Unix platforms, other bits than those listed above may be set.

- uid** An integer which indicates the owner of the file. Ignored for non-Unix file systems.
- gid** An integer which gives the group that the owner of the file belongs to. Ignored non-Unix file systems.

Typical error reasons:

- eaccess** 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

eaaces 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
eival invalid argument
eio I/O error
eisdir illegal operation on a directory
eloop too many levels of symbolic links
emfile too many open files
mlink 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 to serve other processes during the file operation. This gives the same effect as when using the thread pool, but with larger overhead. Other file operations, for example `sync/1` or `close/1` on a huge file, still is a problem.

For increased performance, raw files are recommended. Raw files, uses the file system of the node's host machine. For normal files (non-raw) the file server is used to find the files, and if the node is running its file server as slave to another node's, and the other node runs on some other host machine, they may have different file systems. This is seldom a problem, but you have now been warned.

A normal file is really a process so it can be used as an I/O device (see `io`). Therefore when data is written to a normal file, the sending of the data to the file process copies all data that is not binaries. Opening the file in binary mode and writing binaries is therefore recommended. If the file is opened on another node, or if the file server runs as slave to another node's, also binaries are copied.

Caching data to reduce the number of file operations, or rather the number of calls to the file driver, will generally increase performance. The following function writes 4 MBytes in 23 seconds on my machine:

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

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

The following 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 =< N ->
    create_file(FD, M, M + 1024, []),
    create_file(FD, M + 1024, N);
create_file(FD, M, N) ->
    create_file(FD, M, N, []).
```

```

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

```

Note:

Trust only your own benchmarks. If the list length in `create_file/2` above is increased, it will run slightly faster, but consume more memory and cause more memory fragmentation. How much this affects your application is something that this simple benchmark can not predict.

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

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

A raw file is really a port. When writing data to a port, it is efficient to write a list of binaries. There is no need to flatten a deep list before writing. On Unix hosts, scatter output, which writes a set of buffers in one operation, is used when possible. In this way `file:write(FD, [Bin1, Bin2 | Bin3])` will write the contents of the binaries without copying the data at all except for perhaps deep down in the operating system kernel.

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

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

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

Warnings

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

See Also

`filename(3)`

gen_tcp

Erlang Module

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

The available options are described in the `setopts/2` [page 152] function in the `inet` manual page.

The possible `{error, Reason}` results are described in the `inet` [page 153] manual page.

The following code fragment provides a simple example of a client connecting to a server at port 5678, transferring a binary and closing the connection.

```
client() ->
    SomeHostInNet = "localhost" % to make it runnable on one machine
    {ok, Sock} = gen_tcp:connect(SomeHostInNet, 5678,
                                [binary, {packet, 0}]),
    ok = gen_tcp:send(Sock, "Some Data"),
    ok = gen_tcp:close(Sock).
```

At the other end a server is listening on port 5678, accepts the connection and receives the binary.

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

do_recv(Sock, Bs) ->
    case gen_tcp:recv(Sock, 0) of
        {ok, B} ->
            do_recv(Sock, [Bs, B]);
        {error, closed} ->
            {ok, list_to_binary(Bs)}
    end.
```

Exports

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

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

Types:

- ListenSocket = socket()
- Socket = socket()
- Timeout = integer()
- Reason = atom()

Accepts an incoming connection request on a listen socket. Socket must be a socket returned from `listen/1`. If no Timeout argument is specified, or it is infinity, the `accept` function will not return until a connection has been established or the ListenSocket has been closed. If `accept` returns because the ListenSocket has been closed `{error, closed}` is returned. If Timeout is specified and no connection is accepted within the given time, `accept` will return `{error, timeout}`.

Packets can be sent to the returned socket using the `send/2` function. Packets sent from the peer will be delivered as messages

`{tcp, Socket, Data}`

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

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

Types:

- Socket = socket()
- Reason = atom()

Closes an TCP socket.

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

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

Types:

- Address = string() | atom() | ip_address()
- Port = Timeout = integer()
- Options = list()
- Socket = socket()
- Reason = atom()

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

The available options are:

`list` Received Packet is delivered as a list.

`binary` Received Packet is delivered as a binary.

common inet options The common inet options available are described in the `setopts/2` [page 152] function in the `inet` manual page.

Packets can be sent to the returned socket using the `send/2` function. Packets sent from the peer will be delivered as messages

`{tcp, Socket, Data}`

If the socket was closed the following message is delivered:

`{tcp_closed, Socket}`

If an error occurred on the socket the following message is delivered:

`{tcp_error, Socket, Reason}`

unless the socket is in passive mode, in which case packets are retrieved by calling `recv/2`.

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

`controlling_process(Socket, NewOwner) -> ok | {error, eperm}`

Types:

- `Socket = socket()`
- `NewOwner = pid()`

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

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

Types:

- `Port = integer()`
- `Options = list()`
- `Socket = socket()`
- `Reason = atom()`

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

If the port number is zero, the `listen` function picks an available port number (use `inet:port/1` to retrieve it); otherwise, the specified port number is used.

The available options are described in the `setopts/2` [page 152] function in the `inet` manual page. Additionally, the option `{backlog, B}` can be given, where `B` is an integer ≥ 0 . The backlog value defaults to 5. The backlog value defines the maximum length the queue of pending connections may grow to.

The returned socket can only be used in calls to `accept`.

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

`recv(Socket, Length, Timeout)`

Types:

- `Socket = socket()`
- `Length = integer()`
- `Packet = list() | binary()`

- Timeout = integer()
- Reason = atom()

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

The Length argument is only meaningful when the socket is in raw mode and denotes number of bytes to read. If Length = 0 all available bytes are returned.

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

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

Types:

- Socket = socket()
- Packet = list() | binary()
- Reason = atom()

Sends a packet on a socket.

gen_udp

Erlang Module

The `gen_udp` module is an interface to User Datagram Protocol (UDP).

The possible `{error, Reason}` results are described in the `inet` [page 153] manual page.

Exports

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

Types:

- `Socket = Reason = term()`

Removes the `Socket` created with `open/1` or `open/2`.

`controlling_process(Socket, NewOwner) ->`

Types:

- `Socket = term()`
- `NewOwner = pid()`

The process associated with a `Socket` is changed to `NewOwner`. The `NewOwner` will receive all subsequent data.

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

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

Types:

- `Port = integer(0..65535)`
- `Options = list()`
- `Socket = term()`
- `Reason = term()`

Associates a UDP port number (`Port`) with the calling process. It returns `{ok, Socket}`, or `{error, Reason}`. The returned `Socket` is used to send packets from this port with the `send/4` function. `Options` is a list of options associated with this port.

When UDP packets arrive at the opened `Port` they will be delivered as messages of the type `{udp, Socket, IP, InPortNo, Packet}`. Note that arriving UDP packets that are longer than the receive buffer option specifies might be truncated without warning.

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

The available options are:

`list` Received `Packet` is delivered as a list.

binary Received Packet is delivered as a binary.

common inet options The common inet options available are described in the `setopts/2` [page 152] function in the `inet` manual page. Default value for the receive buffer option is `{recbuf, 8192}`.

If you set `Port` to 0, the underlying Operating System assigns a free UDP port. (You can find out which port by calling `inet:port(Socket)`.)

If any of the following functions are called with a `Socket` that was not opened by the calling process, they will return `{error, not_owner}`. The ownership of a `Socket` can be transferred to another process with `controlling_process/2`.

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

Types:

- `Socket` = `socket()`
- `Address` = `{ integer(), integer(), integer(), integer() }`
- `Port` = `integer(0..65535)`
- `Length` = `integer()`
- `Packet` = `list() | binary()`
- `Timeout` = `integer()`
- `Reason` = `atom()`

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

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

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

Types:

- `Address` = `{ integer(), integer(), integer(), integer() } | atom() | string()`
- `Port` = `integer(0..65535)`
- `Packet` = `[byte()] | binary()`
- `Reason` = `term()`

Sends `Packet` to the specified address (`address`, `port`). It returns `ok`, or `{error, Reason}`. `Address` can be an IP address expressed as a tuple, for example `{192, 0, 0, 1}`. It can also be a host name expressed as an atom or a string, for example `'somehost.some.domain'`. `Port` is an integer, and `Packet` is either a list of bytes or a binary.

global

Erlang Module

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

1. Registration of global names
2. Global locks
3. Monitoring nodes
4. Maintenance of the fully connected network

These services are controlled via the process `global` which exists on every node. `global` is started automatically when a node is started.

The ability to globally register names is a central concept in the programming of distributed Erlang systems. In this module, the equivalent of the `register/2` and `whereis/1` BIFs are implemented, but for a network of Erlang nodes. A registered name is an alias for a process identity `Pid`. The system monitors globally registered Pids. If a process terminates, the name will also be globally unregistered.

The registered names are stored in replica global name tables on every node. There is no central storage point. Thus, the translation of a name to a `Pid` is extremely quick because it is never a network operation. When any action is taken which results in a change to the global name table all tables on other nodes are automatically updated.

Global locks have lock identities and are set on a specific resource. For instance, the specified resource could be a `Pid` of a process. When a global lock is set access to the locked resource is denied for all other resources other than the lock requester.

Both the registration and lock functionalities are atomic. All nodes involved in these actions will have the same view of the information.

The server also performs the critical task of continuously monitoring changes in node configuration, if a node which runs a globally registered process goes down, the name will be globally unregistered. The server will also maintain a fully connected network. For example, if node `N1` connects to node `N2` (which is already connected to `N3`), the `global` server on `N1` then `N3` will make sure that also `N1` and `N3` are connected. If this is not desired, the command line flag `-connect_all false` must be passed to `init` at boot time. In this case, the name registration facility cannot be used (but the lock mechanism will still work.)

Exports

`del_lock(Id)`

`del_lock(Id, Nodes) -> void()`

Types:

- `Id = {ResourceId, LockRequesterId}`
- `ResourceId = term()`
- `LockRequesterId = term()`
- `Nodes = [node()]`

Deletes the lock `Id` synchronously.

`notify_all_name(Name, Pid1, Pid2) -> none`

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

`random_exit_name(Name, Pid1, Pid2) -> Pid1 | Pid2`

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

`random_notify_name(Name, Pid1, Pid2) -> Pid1 | Pid2`

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

`register_name(Name, Pid)`

`register_name(Name, Pid, Resolve) -> yes | no`

Types:

- `Name = term()`
- `Pid = Pid()`
- `Resolve = {M, F}` where `M:F(Name, Pid, Pid2) -> Pid | Pid2 | none`

Globally notifies all nodes of a new global name in a network of Erlang nodes.

When new nodes are added to the network, they are informed of the globally registered names that already exist. The network is also informed of any global names in newly connected nodes. If any name clashes are discovered, the `Resolve` function is called. Its purpose is to decide which Pid is correct. This function blocks the global name server during its execution. If the function crashes, or returns anything other than one of the Pids, the name is unregistered. This function is called once for each name clash.

There are three pre-defined resolve functions, `random_exit_name`, `random_notify_name` and `notify_all_name`. If no `Resolve` function is defined, `random_exit_name` is used. This means that one of the two registered processes will be selected as correct while the other is killed.

This function is completely synchronous. This means that when this function returns, the name is either registered on all nodes or none.

The function returns `yes` if successful, `no` if it fails. For example, `no` is returned if an attempt is made to register a process with a name that is already in use.

If a process with a registered name dies, or the node goes down, the name is unregistered on all nodes.

```
registered_names() -> [Name]
```

Types:

- Name = term()

Returns a lists of all globally registered names.

```
re_register_name(Name, Pid)
```

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

Types:

- Name = term()
- Pid = Pid()
- Resolve = {M, F} where M:F(Name, Pid, Pid2) -> Pid | Pid2 | none

Atomically changes the registered name Name on all nodes to refer to Pid.

The Resolve function has the same behavior as in `register_name`.

```
send(Name, Msg) -> Pid
```

Types:

- Name = term()
- Msg = term()
- Pid = Pid()

Sends the message Msg to the globally registered process Name. If Name is not a globally registered name, the calling function will exit with reason {badarg, {Name, Msg}}.

```
set_lock(Id)
```

```
set_lock(Id, Nodes)
```

```
set_lock(Id, Nodes, Retries) -> boolean()
```

Types:

- Id = {ResourceId, LockRequesterId}
- ResourceId = term()
- LockRequesterId = term()
- Nodes = [node()]
- Retries = int() > 0 | infinity

Sets a lock on the specified nodes (or on all nodes if none are specified) on `ResourceId` for `LockRequesterId`. If a lock already exists on `ResourceId` for another requester than `LockRequesterId`, and `Retries` is not equal to 0, the process sleeps for a while and will try to execute the action later. When `Retries` attempts have been made, `false` is returned, otherwise `true`. If `Retries` is infinity, `true` is eventually returned (unless the lock is never released).

If no value for `Retries` is given, infinity is used.

This function is completely synchronous.

If a process which holds a lock dies, or the node goes down, the locks held by the process are deleted.

`global` keeps track of all processes sharing the same lock, i.e. if two processes set the same lock both processes must delete the lock.

This function does not address the problem of a deadlock. A deadlock can never occur as long as processes only lock one resource at a time. But if some processes try to lock two or more resources, a deadlock may occur. It is up to the application to detect and rectify a deadlock.

`start()`

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

This function starts the global name server. Normally, the server is started automatically.

`stop() -> void()`

Stops the global name server.

`sync() -> void()`

Synchronizes the global name server with all nodes known to this node. These are the nodes which are returned from `erlang:nodes()`. When this function returns, the global name server will receive global information from all nodes. This function can be called when new nodes are added to the network.

`trans(Id, Fun)`

`trans(Id, Fun, Nodes)`

`trans(Id, Fun, Nodes, Retries) -> Res | aborted`

Types:

- `Id = {ResourceId, LockRequesterId}`
- `ResourceId = term()`
- `LockRequesterId = term()`
- `Fun = fun() | {M, F}`
- `Nodes = [node()]`
- `Retries = int() > 0 | infinity`
- `Res = term()`

Sets a lock on `Id` (using `set_lock/3`). Evaluates `Res = Fun()` if successfully locked and returns `Res`. Returns `aborted` if the lock attempt failed. If `Retries` is set to infinity, the transaction will not abort.

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

```
unregister_name(Name) -> void()
```

Types:

- Name = term()

Globally removes Name from the network of Erlang nodes.

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

Types:

- Name = term()

Returns either an atom `undefined`, or a `Pid` which is globally associated with Name.

global_group

Erlang Module

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

The main advantage of dividing systems to global groups is that the background load decreases while the number of nodes to be updated is reduced when manipulating globally registered names.

The `global_groups`-key in the `.config` file defines the global groups:
`{global_groups, [GroupTuple]}`

Types:

- `GroupTuple = {GroupName, [Node]} | {GroupName, PublishType, [Node]}`
- `GroupName = atom()` (naming a global group)
- `PublishType = normal | hidden`
- `Node = atom()` (naming a node)

A `GroupTuple` without `PublishType` is the same as a `GroupTuple` with `PublishType` equal to `normal`.

The command `erl -config File` starts a node with a configuration file named `File.config`. If the `global_groups`-key is not defined the system will start as a whole, without any partitions. When the key is not defined, the services of this function will not give any extra value to `global(3)`.

A hidden node will establish hidden connections to nodes not part of the same global group, and normal (visible) connections to nodes part of the same global group. Hidden connections aren't published on neither of the connected nodes, i.e. neither of the connected nodes are part of the result from `nodes/0` on the other node.

In a hidden global group (a global group defined with `PublishType` equal to `hidden`) all nodes are hidden nodes.

Hidden nodes can also be part of normal global groups. Nodes started with the `-hidden` switch will be hidden nodes even if they are part of a normal group, see `erl(1)`. Other nodes in the group will not be affected by this.

For the processes and nodes to run smoothly using this function the following criteria must be met:

- The global group function must have a server process, `global_group`, running on each node.
NOTE: The processes are automatically started and synchronized when a node is started.

- All processes must agree with the group definition in the immediate global group. If two nodes do not agree, these nodes will not synchronize their name space and an error message will be logged in the error logger.
Example: If one node has an illegal global group definition, such a node will run isolated from the other nodes regarding the global name space; but not regarding other system functions, e.g distribution of applications, refer to chapter NOTE below.
- Nodes can only belong to one global group.

When the global group definitions are to be changed in a system upgrade, the upgrade must fulfill the following steps:

1. First, all nodes which are to be removed from a global group must be taken down.
2. Nodes which are not affected by the redefinition of the global groups are to be upgraded to be aware of the new global group definitions.
NOTE: All nodes in the system, also nodes in unchanged global groups, must be upgraded. This because e.g `send` must have an accurate view of the total system.
3. Finally, all nodes which are new to a global group can be started.

When a non partitioned system is to be upgraded to become a partitioned system, all nodes belonging to a global group will be disconnected from all nodes not belonging to its immediate global group.

Exports

```
global_groups() -> {OwnGroupName, [OtherGroupName]} | undefined
```

Types:

- OwnGroupName = atom()
- OtherGroupName = atom()
- ErrorMsg = term()

Returns the names of all the global groups known to the immediate global group.

```
info() -> [{state, State}, {own_group_name, atom()}, {own_group_nodes, [Node]},  
          {synced_nodes, [Node]}, {sync_error, [Node]}, {no_contact, [Node]},  
          {other_groups, Other_grps}, {monitoring, [pid()]}]
```

Types:

- State = no_conf | synced
- Other_grps = [{OtherGrpName, [Node]}]
- OtherGrpName = atom()
- Node = atom()

Returns the state of the global group process. In the following 'nodes' refers to nodes in the immediate global group. `synced_nodes` lists the nodes this node is synchronized with at this moment. `lists` the nodes defining the own global group. `sync_error` lists the nodes with this node could not be synchronize. `no_contact` lists nodes with this node do not yet have established contact. `other_groups` shows the definition of the other global groups in the system. `monitoring` lists the processes which have subscribed on `nodeup` and `nodedown` messages.

`monitor_nodes(Flag) -> ok`

Types:

- `Flag = bool()`

The requesting process receives `{nodeup, Node}` and `{nodedown, Node}` messages about the nodes from the immediate global group. If the flag `Flag` is set to `true` the service is turned on; `false` turns it off.

`own_nodes() -> [Node] | {error, ErrorMsg}`

Types:

- `Node = atom()`
- `ErrorMsg = term()`

Returns the names of all nodes from the immediate global group, despite of the status of the nodes. Use `info/0` to get the information of the current status of the nodes.

`registered_names({node, Node}) -> [Name] | {error, ErrorMsg}`

`registered_names({group, GlobalGroupName}) -> [Name]`

Types:

- `Name = term()`
- `Node = atom()`
- `GlobalGroupName = atom()`
- `ErrorMsg = term()`

Returns a lists of all globally registered names on the specified node or from the specified global group.

`send(Name, Msg) -> Pid | {badarg, Msg} | {error, ErrorMsg}`

`send({node, Node}, Name, Msg) -> Pid | {badarg, Msg} | {error, ErrorMsg}`

`send({group, GlobalGroupName}, Name, Msg) -> Pid | {badarg, Msg} | {error, ErrorMsg}`

Types:

- `GlobalGroupName = atom()`
- `Msg = term()`
- `Name = term()`
- `Node = atom()`
- `Pid = pid()`
- `ErrorMsg = term()`

`send/2` searches for the registered `Name` in all global groups defined, in the order of appearance in the `.config`-file, until the registered name is found or all groups are searched. If `Name` is found, the message `Msg` is sent to it. If it is not found, the function exits with reason `{badarg, {Name, Msg}}`.

`send/3` searches for the registered `Name` in either the specified node or the specified global group. If the registered name is found, the message `Msg` is sent to that process. If `Name` is not found, the function exits with reason `{badarg, {Name, Msg}}`.

`sync() -> ok`

`sync` synchronizes the global name servers on the nodes in the immediate global group. It also unregisters the names registered in the immediate global group on known nodes outside to the immediate global group.

If the `global_groups` definition is invalid, the function exits with reason `{error, {'invalid global_groups definition', NodeGrpDef}}`.

```
whereis_name(Name) -> Pid | undefined | {error, ErrorMsg}
```

```
whereis_name({node, Node}, Name) -> Pid | undefined | {error, ErrorMsg}
```

```
whereis_name({group, GlobalGroupName}, Name) -> Pid | undefined | {error, ErrorMsg}
```

Types:

- `GlobalGroupName` = `atom()`
- `Name` = `term()`
- `Node` = `atom()`
- `Pid` = `pid()`

`whereis_name/1` searches for the registered `Name` in all global groups defined, in the order of appearance in the `.config`-file, until the registered name is found or all groups are searched.

`whereis_name/2` searches for the registered `Name` in either the specified node or the specified global group.

Returns either the atom `undefined`, or the `Pid` which is associated with `Name`.

```
start()
```

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

This function starts the global group server. Normally, the server is started automatically.

```
stop() -> void()
```

Stop the global group server.

NOTE

In the situation where a node has lost its connections to other nodes in its global group but has connections to nodes in other global groups, a request from the other global group may produce an incorrect or misleading result. When this occurs the isolated node may not have accurate information, for example, about registered names in its global group.

Note also that the `send` function is not secure.

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

SEE ALSO

`erl(1)`, `global(3)`

heart

Erlang Module

The `heart` module sends periodic heartbeats to an external port program, which is also named `heart`. The purpose of the `heart` port program is to check that the Erlang runtime system it is supervising is still running. If the port program has not received any heartbeats within `HEART_BEAT_TIMEOUT` (default is 60 seconds) from the last one, the system can be rebooted. Also, if the system is equipped with a hardware watchdog timer and is running Solaris, the watchdog can be used to supervise the entire system.

This module is started by the `init` module during system start-up. The `-heart` command line flag determines if the `heart` module should start.

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

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

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

The `HEART_BEAT_TIMEOUT` and `HEART_BEAT_BOOT_DELAY` environment variables can be used to configure the heart timeouts, they can be set in the operating system shell before `erl -heart` is started or can be passed on the command line like this: `erl -heart -env HEART_BEAT_TIMEOUT 30`.

The value (in seconds) must be in the range $10 < X \leq 65535$.

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

Exports

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

Types:

- `Pid = pid()`
- `What = void()`

Starts the `heart` program. This function returns `ignore` if the `-heart` command line flag is not supplied.

`set_cmd(Cmd) -> ok | {error, {bad_cmd, Cmd}}`

Types:

- `Cmd = string()`

Sets a temporary reboot command. This command is used if a `HEART_COMMAND` other than the one specified with the environment variable should be used in order to reboot the system. The new Erlang runtime system will (if it misbehaves) use the environment variable `HEART_COMMAND` to reboot.

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

`clear_cmd() -> ok`

Clears the temporary boot command. If the system terminates, the normal `HEART_COMMAND` is used to reboot.

inet

Erlang Module

Inet provides access to TCP/IP protocols.

Some functions returns a `hostent` record. Use this line in your module

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

to include the record definition.

`h_addr_list` List of addresses for this host

`h_addrtype` Type of address: `inet` or `inet6`

`h_aliases` List of aliases (additional names for host)

`h_length` Length of address in bytes

`h_name` Official name for host

Addresses as inputs to functions can be either a string or a tuple. For instance, the IP address 150.236.20.73 can be passed to `gethostbyaddr/1` either as the string "150.236.20.73" or as the tuple `{150, 236, 20, 73}`. Addresses returned by any function in the `inet` module will be a tuple.

Hostnames may be specified as either atoms or a strings.

Where an address family is required, valid values are `inet` (standard IPv4 addresses) or `inet6` (IPv6).

Exports

`format_error(Tag)`

Types:

- `Tag = atom()`

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

`gethostbyaddr(Address) -> {ok, Hostent} | {error, Reason}`

Types:

- `Address = address()`
- `Hostent = hostent()`

Returns a `hostent` record given an address.

`gethostbyname(Name) -> {ok, Hostent} | {error, Reason}`

Types:

- Hostname = hostname()
- Hostent = hostent()

Returns a hostent record given a hostname.

gethostbyname(Name, Family) -> {ok, Hostent} | {error, Reason}

Types:

- Hostname = hostname()
- Family = family()
- Hostent = hostent()

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

gethostname() -> {ok, Name} | {error, Reason}

Types:

- Hostname = hostname()

Returns the local hostname. Will never fail.

sockname(Socket) -> {ok, {IP, Port}} | {error, Reason}

Types:

- Socket = socket()
- Address = address()
- Port = integer()

Returns the local address and port number for a socket.

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

Types:

- Socket = socket()
- Address = address()
- Port = integer()

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

port(Socket) -> {ok, Number}

Types:

- Socket = socket()
- Number = integer()

Returns the local port number for a socket.

close(Socket) -> ok

Types:

- Socket = socket()

Closes a socket of any type.

```
getaddr(IP,inet) -> {ok,{A1,A2,A3,A4}} | {error, Reason}
```

Types:

- IP = {A1,A2,A3,A4} | string() | atom()
- A1 = A2 = A3 = A4 = integer()
- Reason = term()

Returns the IP-address as a tuple with integers for IP which can be an IP-address a single hostname or a fully qualified hostname. At present only IPv4 addresses (the `inet` argument) is supported, but the function is prepared to support IPv6 (`inet6`) in a near future.

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

Types:

- Socket = term()
- Options = list()

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

`{active, Boolean}` If the active option is `true`, which is the default, everything received from the socket will be sent as messages to the receiving process. If the active option is set to `false` (passive mode), the process must explicitly receive incoming data by calling `gen_tcp:recv/N` or `gen_udp:recv/N` (depending on the type of socket). Note: Passive mode provides flow control; the other side will not be able send faster than the receiver can read. 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.

`{broadcast, Boolean}` Enable/disable permission to send broadcasts. (UDP)

`{dontroute, true|false}` Use `{dontroute, true}` to enable/disable routing bypass for outgoing messages.

`{header, Size}` This option is only meaningful if the binary option was specified when the socket was created. If the header option is specified, the first `Size` number bytes of data received from the socket will be elements of a list, and the rest of the data will be a binary given as the tail of the same list. If for example `Size=2` the data received will match `[Byte1,Byte2|Binary]`.

`{keepalive, Boolean}` (TCP/IP sockets) Enables periodic transmission on a connected socket, when no other data is being exchanged. If the other end does not respond, the connection is considered broken and an error message will be sent to the controlling process. Default disabled.

`{nodelay, Boolean}` If `Boolean` is `true`, the `TCP_NODELAY` option is turned on for the socket, which means that even small amounts of data will be sent immediately. (TCP/IP sockets)

`{packet, PacketType}` (TCP/IP sockets) Defines the type of packets to use for a socket. The following values are valid:

`raw` | `0` No packaging is done.

- 1 | 2 | 4 Packets consist of a header specifying the number of bytes in the packet, followed by that number of bytes. The length of header can be one, two, or four bytes; the order of the bytes is big-endian. Each send operation will generate the header, and the header will be stripped off on each receive operation.
- asn1 | cdr | sunrm | fcgi | tpkt | line These packet types only have effect on receiving. When sending a packet, it is the responsibility of the application to supply a correct header. On receiving, however, there will be one message sent to the controlling process for each complete packet received, and, similarly, each call to `gen_tcp:recv/N` returns one complete packet. The header is *not* stripped off. The meanings of the packet types are as follows:
- asn1 - ASN.1 BER,
 - sunrm - Sun's RPC encoding,
 - cdr - CORBA (GIOP 1.1),
 - fcgi - Fast CGI,
 - tpkt - TPKT format [RFC1006],
 - line - Line mode, a packet is a line terminated with newline, lines longer than the receive buffer are truncated.
- {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.

ERRORS

The possible error reasons and the corresponding diagnostic strings returned by `format_error/1` are as follows:

e2big argument list too long
 eaccess 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
eival invalid argument
eio I/O error
eisconn socket is already connected
eisdir illegal operation on a directory
eisnam is a named file
el2hlt level 2 halted
el2nsync level 2 not synchronized
el3hlt level 3 halted
el3rst level 3 reset
elbin ELBIN
elibacc cannot access a needed shared library
elibbad accessing a corrupted shared library
elibexec cannot exec a shared library directly
elibmax attempting to link in more shared libraries than system limit
elibscn .lib section in a.out corrupted

elnrng link number out of range
eloop too many levels of symbolic links
emfile too many open files
emlink too many links
emsgsize message too long
emultihop multihop attempted
enametoolong file name too long
enavail not available
enet ENET
enetdown network is down
enetreset network dropped connection on reset
enetunreach network is unreachable
enfile file table overflow
enoano anode table overflow
enobufs no buffer space available
enocsi no CSI structure available
enodata no data available
enodev no such device
enoent no such file or directory
enoexec exec format error
enolck no locks available
enolink link has be severed
enomem not enough memory
enomsg no message of desired type
enonet machine is not on the network
enopkg package not installed
enoprotoopt bad proocol option
enospc no space left on device
enosr out of stream resources or not a stream device
enosym unresolved symbol name
enosys function not implemented
enotblk block device required
enotconn socket is not connected
enotdir not a directory
enotempty directory not empty
enotnam not a named file
enotsock socket operation on non-socket
enotsup operation not supported
enotty inappropriate device for ioctl
enotuniq name not unique on network
enxio no such device or address

eopnotsupp operation not supported on socket
eperm not owner
epfnosupport protocol family not supported
epipe broken pipe
eproclim too many processes
eprocunavail bad procedure for program
eprogmismatch program version wrong
eprogunavail RPC program not available
eproto protocol error
eprotonosupport protocol not supported
eprototype protocol wrong type for socket
erange math result unrepresentable
erefused EREFUSED
eremchg remote address changed
eremdev remote device
eremote pathname hit remote file system
eremoteio remote i/o error
eremoterelease EREMOTERELEASE
erofs read-only file system
erpcmismatch RPC version is wrong
erremote object is remote
eshutdown cannot send after socket shutdown
esocktnosupport socket type not supported
espipe invalid seek
esrch no such process
esrmnt srmount error
estale stale remote file handle
esuccess Error 0
etime timer expired
etimedout connection timed out
etoomanyrefs too many references
etxtbsy text file or pseudo-device busy
euclean structure needs cleaning
eunatch protocol driver not attached
eusers too many users
eversion version mismatch
ewouldblock operation would block
exdev cross-domain link
exfull message tables full
nxdomain the hostname or domain name could not be found

init

Erlang Module

`init` is pre-loaded into the system before the system starts and it coordinates the start-up of the system. The first function evaluated at start-up is `boot(Bootargs)`, where `Bootargs` is a list of the arguments supplied to the Erlang runtime system from the local operating system. The Erlang code for the module `init` is always pre-loaded.

`init` reads a boot script which contains instructions on how to initiate the system. The default boot script (`start.boot`) is in the directory `<ERL_INSTALL_DIR>/bin`.

`init` contains functions to fetch command line flags, or arguments, supplied to the Erlang runtime system.

`init` also contains functions to restart, reboot, and stop the system.

Exports

`boot(BootArgs) -> void()`

Types:

- `BootArgs = [binary()]`

Erlang is started with the command `erl <script-flags> <user-flags>`.

`erl` is the name of the Erlang start-up script. `<script-flags>`, described in `erl(1)`, are read by the script. `<user-flags>` are put into a list and passed as `Args` to `boot/1`.

The `boot/1` function interprets the `boot`, `mode`, and `s` flags. These are described in `COMMAND LINE FLAGS`.

If the `boot` function finds other arguments starting with the character `-`, that argument is interpreted as a flag with zero or more values. It ends the previous argument. For example:

```
erl -run foo bar -charles peterson
```

This starts the Erlang runtime system, evaluates `foo:bar()`, and sets the flag `-charles`, which has the associated value `peterson`.

Other arguments which are passed to the `boot` function, and do not fit into the above description, are passed to the `init` loop as plain arguments.

The special flag `--` can be used to separate plain arguments to `boot` from a preceding flag argument.

The special flag `-extra` causes all following arguments to become plain arguments, and not be subjected to any interpretation by Erlang.

`get_arguments() -> Flags`

Types:

- Flags = [{Flag,FValue}]
- Flag = atom()
- FValue = [Value]
- Value = string()

Returns all flags given to the system.

`get_argument(Flag) -> {ok, Values} | error`

Types:

- Flag = atom()
- Values = [FValue]
- FValue = [Value]
- Value = string()

Returns all values associated with Flag. If Flag is provided several times, each FValue is returned in preserved order.

`get_args() -> [Arg]`

Types:

- Arg = atom()

Returns the additional plain arguments as a list of atoms (possibly empty). It is recommended that `get_plain_arguments/1` be used instead, because of the limited length of atoms.

`get_plain_arguments() -> [Arg]`

Types:

- Arg = string()

Returns the additional plain arguments as a list of strings (possibly empty).

`restart() -> void()`

The system is restarted inside the running Erlang node, which means that the emulator is not restarted. All applications are taken down smoothly, all code is unloaded, and all ports are closed before the system is booted again in the same way as initially started. The same `BootArgs` are used again.

To limit the shutdown time, the time `init` is allowed to spend taking down applications, the `-shutdown_time` command line flag should be used.

`reboot() -> void()`

All applications are taken down smoothly, all code is unloaded, and all ports are closed before the Erlang node terminates. If the `-heart` system flag was given, the `heart` program will try to reboot the system. Refer to the `heart` module for more information.

In order to limit the shutdown time, the time `init` is allowed to spend taking down applications, the `-shutdown_time` command line flag should be used.

`stop() -> void()`

All applications are taken down smoothly, all code is unloaded, and all ports are closed before the system terminates. If the `-heart` system flag was given, the heart program is terminated before the Erlang node terminates. Refer to the heart module for more information.

In order to limit the shutdown time, the time init is allowed to spend taking down applications, the `-shutdown_time` command line flag should be used.

`get_status()` -> {InternalStatus, ProvidedStatus}

Types:

- InternalStatus = starting | started | stopping
- ProvidedStatus = term()

The current status of the `init` process can be inspected. During system start (initialization), InternalStatus is `starting`, and ProvidedStatus indicates how long the boot script has been interpreted. Each {progress, Info} term interpreted in the boot script affects the ProvidedStatus status, i.e., ProvidedStatus gets the value of Info.

`script_id()` -> Id

Types:

- Id = term()

Get the identity of the boot script used to boot the system. Id can be any Erlang term. In the delivered boot scripts, Id is {Name, Vsn}. Name and Vsn are strings.

Command Line Flags

The `init` module interprets the following flags:

-boot File Specifies the name of the boot script, File.boot, used to start the system. Unless File contains an absolute path, the system searches for File.boot in the current and <ERL_INSTALL_DIR>/bin directories

If this flag is omitted, the <ERL_INSTALL_DIR>/bin/start.boot boot script is used.

-boot_var Var Directory [Var Directory] If the boot script used contains another path variable than \$ROOT, that variable must have a value assigned in order to start the system. A boot variable is used if user applications are installed in a different location than underneath the <ERL_INSTALL_DIR>/lib directory. \$Var is expanded to Directory in the boot script.

-mode Mode The mode flag indicates if the system will load code automatically at runtime, or if all code should be loaded during system initialization. Mode can be either `interactive` (allow automatic code loading) or `embedded` (load all code during start-up).

-shutdown_time Time Specifies how long time (in ms) the `init` process is allowed to spend shutting down the system. If Time milliseconds has elapsed, all processes still existing are *killed*.

If `-shutdown_time` is not specified, the default time is infinity.

-run Module [Function [Args]] Evaluate the function during system initialization.

Function defaults to `start` and `Args` to `[]`. If the function call ends abnormally, the Erlang runtime system stops with an error message.

The arguments after `-run` are used as arguments to Erlang functions. All arguments are passed as strings. For example:

```
erl -run foo -run foo bar -run foo bar baz 1 2
```

This starts the Erlang runtime system and then evaluates the following Erlang functions:

```
foo:start()
foo:bar()
foo:bar([baz, "1", "2"]).
```

The functions are executed sequentially in the initialization process, which then terminates normally and passes control to the user. This means that a `-run` call which does not terminate will block further processing; to avoid this, use some variant of `spawn` in such cases.

-s Module [Function [Args]] Evaluate the function during system initialization.

Function defaults to `start` and `Args` to `[]`. If the function call ends abnormally, the Erlang runtime system stops with an error message.

The arguments after `-s` are used as arguments to Erlang functions. All arguments are passed as atoms. For example:

```
erl -s foo -s foo bar -s foo bar baz 1 2
```

This starts the Erlang runtime system and then evaluates the following Erlang functions:

```
foo:start()
foo:bar()
foo:bar([baz, '1', '2']).
```

The functions are executed sequentially in the initialization process, which then terminates normally and passes control to the user. This means that a `-s` call which does not terminate will block further processing; to avoid this, use some variant of `spawn` in such cases.

Due to the 255 character limit on atoms, it is recommended that `-run` be used instead.

-init.debug The `init` process writes some debug information while interpreting the boot script.

Example

```
erl -- a b -children thomas claire -ages 7 3 -- x y
1> init:get_plain_arguments().
["a", "b", "x", "y"]
2> init:get_argument(children).
{ok, [["thomas", "claire"]]}
3> init:get_argument(ages).
```

```
{ok, [{"7", "3"]]}
4> init:get_argument(silly).
error
```

See also

`erl_prim_loader(3)`, `heart(3)`

net_adm

Erlang Module

This module contains various network utility functions.

Exports

`host_file()`

This function reads the `.hosts.erlang` file. It returns the hosts in this file as a list, or it returns `{error, Reason}` if the file cannot be found.

`dns_hostname(Host)`

This function calls `epmd` for the fully qualified name (DNS) of `Host`. It returns `{ok, Longhostname}` if the call is successful, or `{error, Host}` if `Host` cannot be located by DNS.

`localhost()`

This function returns the fully qualified name of the local host, if it can be found by DNS.

`names(), names(Host)`

This function returns `{ok, List}` or `{error, Reason}`. `List` is a list of tuples on the form `{Name, Port}`. For example: `net_adm:names(elrond) -> {ok, [{"foo", 61178}, {"ts", 61160}]}`.

`ping(Node)`

This function tries to set up a connection to `Node`. It returns `pang` if it fails, and `pong` if it is successful.

`world (), world (verbose)`

This function runs `epmd - names` on all hosts which are specified in the Erlang host file `.hosts.erlang`, collects the replies and then evaluates `ping` on all those nodes. Accordingly, connections are created to all nodes which are running on the hosts specified in the `.hosts.erlang` file. An error message is printed if another user node is found when this is done.

This function can be useful when a node is started, but the names of the other nodes in the network are not initially known.

`world_list (Hostlist), world_list (Hostlist, verbose)`

These functions are the same as `world/0` and `world/1`, but instead of reading the hostfile from `.hosts.erlang` the hosts are specified in `Hostlist`.

Files

The `.hosts.erlang` file consists of a number of host names written as Erlang terms. It can be located in the current work directory, `$HOME/.hosts.erlang`, or `code:root_dir()/.hosts.erlang`. The format of the `.hosts.erlang` file must be one host name per line. The host names must be within quotes as shown in the following examples:

```
'super.eua.ericsson.se'.  
'renat.eua.ericsson.se'.  
'grouse.eua.ericsson.se'.  
'gauffin1.eua.ericsson.se'.  
^ (new line)
```

net_kernel

Erlang Module

The net kernel is a system process which must be running for distributed Erlang to work. The purpose of this process is to implement parts of the BIFs `spawn/4` and `spawn_link/4`, and to provide authentication and monitoring of the network.

An Erlang runtime system can be started from the UNIX command line as follows:

```
% erl -name foobar
```

With this command line, the `net_kernel` is started as `net_kernel:start([foobar])`. See `erl(1)`.

This is done by the system itself, but the `start([Name])` function can also be called directly from the normal Erlang shell prompt, and a normal Erlang runtime system is then converted to a node. The kernel can be shut down with the function `stop()`, but only if the kernel was not started by the system itself. The node is then converted into a normal Erlang runtime system. All other nodes on the network will regard this as a total node crash.

If the system is started as `% erl -sname foobar`, the node name of the node will be `foobar@Host`, where `Host` is the short name of the host (not the fully qualified domain name). The `-name` flag gives a node with the fully qualified domain name. See `erl(1)`.

The system can be started with the flag `-dist_auto_connect` to control automatic connection of remote nodes. See `connect_node/1` below and `erl(1)`.

Exports

`kernel_apply(M, F, A)`

As the net kernel runs in the 'user space', it is easy to provide another net kernel which is tailor made for a specific application. For example, the user supplied kernel can limit the set of registered processes which can be accessed from remote nodes, or it can spawn a new process for each `{nodeup, Node}` message and perform some application specific user authentication, a log-in procedure for example. The `kernel_apply(M, F, A)` function is supplied for this purpose.

`monitor_nodes(Flag)`

A process which evaluates this function receives copies of the `{nodeup, Node}` and `{nodedown, Node}` messages that the net kernel receives from the runtime system. The flag `Flag` is set to `true` to turn the service on, and `false` to turn it off.

Nodes connected through hidden connections, see `global_group(3)`, cannot be monitored with `net_kernel:monitor_nodes/1`.

`allow(NodeList)`

In a simple way, this function limits access to a node from a specific number of named nodes. A node which evaluates this function can only be accessed from nodes listed in the `NodeList` variable. Any access attempts made from nodes not listed in `NodeList` are rejected.

`connect_node(Node)`

Explicitly establishes a connection to the node specified by the atom `Node`. Returns `true` if successful, `false` if not, and `ignored` if `net_kernel` is not started.

This function is only necessary if the system is started with the flag `-dist_auto_connect`. See `erl(1)`.

OS

Erlang Module

The functions in this module are operating system specific. Careless use of these functions will result in programs that will only run on a specific platform. On the other hand, with careful use these functions can be of help in enabling a program to run on most platforms.

Exports

`cmd(Command) -> string()`

Types:

- `Command = string() | atom()`

Executes `Command` in a command shell of the target OS and returns the result as a string. This function is a replacement of the previous `unix:cmd/1`; on a Unix platform they are equivalent.

Examples:

```
LsOut = os:cmd("ls"), % on unix platform
DirOut = os:cmd("dir"), % on Win32 platform
```

`find_executable(Name) -> Filename | false`

`find_executable(Name, Path) -> Filename | false`

Types:

- `Name = string()`
- `Path = string()`
- `Filename = string()`

These two functions look up an executable program given its name and a search path, in the same way as the underlying operating system. `find_executable/1` uses the current execution path (i.e., the environment variable `PATH` on Unix and Windows). `Path`, if given, should conform to the syntax of execution paths on the operating system. The absolute filename of the executable program `Name` is returned, or `false` if the program was not found.

`getenv() -> List`

Types:

- `List = list() of string`

Returns a list of all environment variables. Each environment variable is a single string, containing the name of the variable, followed by =, followed by the value.

`getenv(VarName) -> Value | false`

Types:

- Varname = string()
- Value = string()

Returns the Value of the environment variable VarName, or false if the environment variable is undefined.

`getpid() -> Value`

Types:

- Value = string()

Returns the process identifier of the current Erlang emulator in the format most commonly used by the operating system environment. Value is returned as a string containing the (usually) numerical identifier for a process. On Unix, this is typically the return value of the `getpid()` system call. On VxWorks, Value contains the task id (decimal notation) of the Erlang task. On Windows, the process id as returned by the `GetCurrentProcessId()` system call is used.

`putenv(VarName, Value) -> true`

Types:

- Varname = string()
- Value = string()

Sets a new Value for the environment variable VarName.

`type() -> {Osfamily,Osname} | Osfamily`

Types:

- Osfamily = atom() = win32 | unix | vxworks
- Osname = atom()

Returns the Osfamily and, in some cases, Osname of the current operating system.

On Unix, Osname will be same string that `uname -s` returns, but in lower case. For instance, on Solaris 1 and 2, the atom `sunos` will be returned.

In Windows, Osname will be either `nt` (on Windows NT), or `windows` (on Windows 95). On VxWorks Osfamily alone is returned, i.e. the atom `vxworks`.

Note:

Think twice before using this function. Use the `filename` module if you want to inspect or build file names in a portable way. Avoid matching on the Osname atom.

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

Types:

- Major = Minor = Release = integer()
- VersionString = string()

Returns the operating system version. On most systems, this function returns a tuple, but a string will be returned instead if the system has versions which cannot be expressed as three numbers.

Note:

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

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 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(3), pg(3)

rpc

Erlang Module

This module contains services which are similar to remote procedure calls. It also contains broadcast facilities and parallel evaluators. A remote procedure call is a method to call a function on a remote node and collect the answer. It is used for collecting information on a remote node, or for running a function with some specific side effects on the remote node.

Exports

`start()`

Starts the `rpc` server. Normally, this is not necessary because the `rpc` server is started automatically.

`stop()`

Stops the `rpc` server.

`call(Node, Module, Function, Args)`

Evaluates `apply(Mod, Fun, Args)` on the node `Node` and returns a value, or `{badrpc, Reason}` if the call fails.

`cast(Node, Module, Function, Args)`

Causes the expression `apply(Mod, Fun, Args)` to be evaluated on `Node`. No response is delivered and the process which makes the call is not suspended until the evaluation is complete, as is the case with `call/4`. The function immediately returns `true`. Example:

```
> rpc:cast(Node, erlang, halt, [])
```

This function shuts down the node `Node`.

The following function also shuts down the node, but the call returns the tuple `{badrpc, noconnection}`

```
> rpc:call(Node, erlang, halt, [])
```

`block_call(Node, Mod, Fun, Args)`

The `call/4` function causes the server at `Node` to create a new process for each request. This means that several RPCs can be active concurrently. The `rpc` server is not affected if a request does not return a value. This function can be used if the intention of the call is to block the `rpc` server from any other incoming requests until the request has been handled. The function can also be used for efficiency reasons when very small fast functions are evaluated, for example BIFs that are guaranteed not to suspend.

```
> rpc:block_call(Node, erlang, whereis, [file_server]),
```

Returns the Pid of the file server at Node.

`server_call(Node, Name, ReplyWrapper, Msg)`

This function is used when interacting with a server called `Name` at node `Node`. It is assumed that the server receives messages in the format `{From, Request}` and replies in the format `From ! {ReplyWrapper, node(), Reply}`. This function makes such a server call and ensures that the entire call is packed into an atomic transaction which either succeeds or fails. It never hangs, unless the server itself hangs.

The function returns `{error, Reason}`, or the answer as produced by the server `Name`.

`abcast(Name, Mess)`

Broadcasts the message `Mess` asynchronously to the registered process `Name` on all nodes, including the current node.

`abcast(Nodes, Name, Mess)`

The same as `abcast/2`, but only to the nodes `Nodes`.

`sbcast(Name, Msg)`

Broadcasts to all nodes synchronously and returns a list of the nodes which have `Name` as a registered server. Returns `{Goodnodes, Badnodes}`.

It is synchronous in the sense that it is known that all servers have received the message when we return from the call. It is not possible to know that the servers have actually processed the message.

Any further messages sent to the servers, after this function has returned, will be received by all servers after this message .

`sbcast(Nodes, Name, Msg)`

As `sbcast/2` but only to the nodes in `Nodes`.

`eval_everywhere(Mod, Fun, Args)`

Evaluates the expression `apply(Mod, Fun, Args)` on all nodes. No answers are collected.

`eval_everywhere(Nodes, Mod, Fun, Args)`

Evaluates the expression `apply(Mod, Fun, Args)` on the nodes `Nodes`. No answers are collected.

`multicall(M, F, A)`

In contrast to an RPC, a multicall is an RPC which is sent concurrently from one client to multiple servers. This is useful for collecting some information from a set of nodes, or for calling a function on a set of nodes to achieve some side effects. It is semantically the same as iteratively making a series of RPCs on all the nodes, but the multicall is faster as all the requests are sent at the same time and are collected one by one as they come back.

The function `multicall/3` evaluates the expression `apply(M, F, A)` on all nodes and collects the answers. It returns `{Replies, Badnodes}`, where `Badnodes` is a list of the nodes that terminated during computation and `Replies` is a list of the return values. This is useful when new object code is to be loaded on all nodes in the network.

```
%% Find object code for module Mod
{Mod, Bin, File} = code:get_object_code(Mod),

%% and load it on all nodes including this one
{Replies, _} = rpc:multicall(code, load_binary, [Mod, Bin, File,]),

%% and then maybe check the Replies list.
```

This is an example of the side effects the RPCs may produce.

```
multicall(Nodes, M, F, A)
```

Executes the multicall only on the nodes `Nodes`.

```
multi_server_call(Name, Msg)
```

The function sends `Msg` to `Name` on all nodes, and collects the answers. It returns `{Replies, Badnodes}`, where `Badnodes` is a list of the nodes which failed during the call. This function assumes that if a request sent to a server called `Name`, the server replies in the form `{Name, node(), Reply}`. Otherwise, the function will hang. It also assumes that the server receives messages in the form `{From, Msg}`, and then replies as `From ! {Name, node(), Reply}`.

If any of the nodes or servers does not exist or crashes during the call, they appear in the `Badnodes` list.

Warning:

If any of the nodes are of an older release of Erlang, the server cannot be monitored, and this function hangs if the server does not exist.

If all nodes are of the current release of Erlang, `safe_multi_server_call/2,3` is now obsolete and much more inefficient than `multi_server_call/2,3`.

The replies are not ordered in any particular way.

```
multi_server_call(Nodes, Name, Msg)
```

The same as above, but `Msg` is only sent to `Nodes`.

```
safe_multi_server_call(Name, Msg)
```

The same as the `multi_server_call/2`, except that this function handles the case where the remote node exists, but no server called `Name` exists there, and the remote node is of an older release of Erlang. This call is also slightly slower than `multi_server_call/2` since all request go via the rpc server at the remote sites.

`safe_multi_server_call(Nodes, Name, Msg)`

The same as above, but only on the nodes `Nodes`.

`async_call(Node, Mod, Fun, Args)`

Call streams with promises is a type of `rpc` which does not suspend the caller until the result is finished. They return a `Key` which can be used at a later stage to collect the value. The key can be viewed as a promise to deliver the answer. The expression `apply(Mod, Fun, Args)` is evaluated for this function on `Node`. Returns `Key` which can be used in a subsequent `yield/1` (see below).

`yield(Key)`

Delivers the promised answer from a previous `async_call` operation. If the answer is available, it is returned immediately. Otherwise, the caller of `yield/1` is suspended until the answer arrives from `Node`.

`nb_yield(Key, Timeout)`

This is a non-blocking version of `yield`. It returns the tuple `{value, V}` when the computation has finished, or the atom `timeout` when `Timeout` elapses.

`Timeout` is either a non-negative integer or the atom `infinity`.

`nb_yield(Key)`

Same as `nb_yield(Key, 0)`.

`parallel_eval(ListOfTuples)`

Evaluates the list of size 3 tuples `ListOfTuples`. Each tuple must be of the type `{Mod, Fun, Args}`. Each tuple is sent for evaluation to neighboring nodes, and the replies are collected and returned as a list of individual values. The return values are presented in the same order as the original list `ListOfTuples`.

`pmap({M, F}, Extraargs, List)`

Takes exactly the same arguments and has the same return value as the `lists:map/3` function, except that everything is evaluated in parallel on different nodes.

`pinfo(Pid)`

Location transparent version of `process_info/1`.

`pinfo(Pid, Item)`

Location transparent version of `process_info/2`.

seq_trace

Erlang Module

Sequential tracing makes it possible to trace all messages resulting from one initial message. Sequential tracing is completely independent of the ordinary tracing in Erlang, which is controlled by the `erlang:trace/3` BIF. See the chapter "What is Sequential Trace" [page 178] below for more information about what sequential tracing is and how it can be used.

`seq_trace` provides functions which control all aspects of sequential tracing. There are functions for activation, deactivation, inspection and for collection of the trace output.

Note:

The implementation of sequential tracing is in beta status. This means that the programming interface still might undergo minor adjustments (possibly incompatible) based on feedback from users.

Exports

`set_token(Component, ComponentValue) -> {Component, PreviousValue}`

Types:

- `Component` = `label` | `serial` | `Flag`
- `Flag` = `send` | `'receive'` | `print` | `timestamp`
- `ComponentValue` = `FlagValue` | `LabelValue` | `SerialValue`
- `FlagValue` = `bool()` (for `Flag`)
- `LabelValue` = `integer()` (for `label`)
- `SerialValue` = `{Previous, Current}`
- `Previous` = `Current` = `integer()`

Sets the individual `Component` of the trace token to `ComponentValue`. Returns the previous value of the trace token `Component`. The valid `Component`, `ComponentValue` combinations are:

`label`, `integer()` The `label` component is an integer which identifies all events belonging to the same sequential trace. If several sequential traces can be active simultaneously `label` is used to identify the separate traces. Default is 0.

`send`, `bool()` A trace token flag (`true` | `false`) which enables/disables tracing on message sending. Default is `false`.

`'receive'`, `bool()` A trace token flag (`true` | `false`) which enables/disables tracing on message reception. Default is `false`.

`print, bool()` A trace token flag (`true` | `false`) which enables/disables tracing on explicit calls to `seq_trace:print/1`. Default is `false`.

`timestamp, bool()` A trace token flag (`true` | `false`) which enables/disables a timestamp to be generated for each traced event. Default is `false`.

`serial, SerialValue` `SerialValue = {Previous, Current}`. The serial component contains counters which enables the traced messages to be sorted, should never be set explicitly by the user as these counters are updated automatically. Default is `{0, 0}`.

`set_token(Token) -> PreviousToken`

Types:

- `Token = PreviousToken = term() | []`

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

```
OldToken = seq_trace:set_token([]), % set to empty and save
                                % old value
% do something that should not be part of the trace
io:format("Exclude the signalling caused by this~n"),
seq_trace:set_token(OldToken), % activate the trace token again
...
```

Returns the previous value of the trace token.

`get_token(Component) -> {Component, ComponentValue}`

Types:

- `Component = label | serial | Flag`
- `ComponentValue = FlagValue | LabelValue | SerialValue`
- `Flag = send | 'receive' | print | timestamp`
- `FlagValue = bool()` (for `Flag`)
- `LabelValue = integer()` (for `label`)
- `SerialValue = {Previous, Current}` (for `serial`)
- `Previous = Current = integer()`

Returns the value of the trace token component `Component`.

`get_token() -> TraceToken`

Types:

- `TraceToken = term() | []`

Returns the value of the trace token for the current process. If `[]` is returned it means that tracing is not active. Any other value returned is the value of an active trace token. The value returned can be used as input to the `set_token/1` function.

`print(TraceInfo) -> void`

Types:

- `TraceInfo = term()`

Puts the Erlang term `TraceInfo` into the sequential trace output if the process currently is executing within a sequential trace and the `print` flag of the trace token is set.

`print(Label, TraceInfo) -> void`

Types:

- `Label = integer()`
- `TraceInfo = term()`

Same as `print/1` with the additional condition that `TraceInfo` is output only if `Label` is equal to the label of the executing process's sequential trace token.

`reset_trace() -> void`

Sets the trace token to empty for all processes in the node. The process internal counters used to create the serial of the trace token is set to 0. The trace token is set to empty for all messages in message queues. Together this will effectively stop all ongoing sequential tracing in the Erlang node.

`set_system_tracer(ProcessOrPortId) -> PreviousId`

Types:

- `Pid = PreviousId = pid() | port() | false`

Sets the system tracer. The system tracer can be either a `pid` or `port` denoted by `ProcessOrPortId`. Returns the previous value (which can be `false` if no system tracer is active). The function will generate `{'EXIT', {badarg, Info}}` if `Pid` is not the `pid` of an existing local process.

`get_system_tracer() -> pid() | port() | false`

Returns the `pid` or `port` identifier of the current system tracer or `false` if no system tracer is activated.

Trace Messages Sent To the System Tracer

The format of the messages are:

`{seq_trace, Label, SeqTraceInfo, TimeStamp}`

or

`{seq_trace, Label, SeqTraceInfo}`

depending on whether the timestamp flag of the trace token is set to `true` or `false`.

Where :

`Label = integer()`

`TimeStamp = {Seconds, Milliseconds, Microseconds}`

`Seconds = Milliseconds = Microseconds = integer()`

The `SeqTraceInfo` can have the following formats:

`{send, Serial, From, To, Message}` Used when a process `From` with its trace token flag `print` set to `true` has sent a message.

```
{'receive', Serial, From, To, Message}  Used when a process To receives a
    message with a trace token that has the 'receive' flag set to true.

{print, Serial, From, _, Info}  Used when a process From has called
    seq_trace:print(Label,Info) and has a trace token with print set to true and
    label set to Label.

Serial = {PreviousSerial, ThisSerial}
PreviousSerial = ThisSerial = integer()
From = To = pid()
```

Serial is a tuple consisting of two integers where the first PreviousSerial denotes the serial counter passed in the last received message which carried a trace token. If the process is the first one in a new sequential trace the PreviousSerial is set to the value of the process internal “trace clock”. The second integer ThisSerial is the serial counter that a process sets on outgoing messages and it is based on the process internal “trace clock” which is incremented by one before it is attached to the trace token in the message.

What is Sequential Tracing

Sequential tracing is a way to trace a sequence of messages sent between different local or distributed processes where the sequence is initiated by one single message. In short it works like this:

Each process has a *trace token* which can be empty or not empty. When not empty the trace token can be seen as the tuple {Label, Flags, Serial, From}. The trace token is passed invisibly with each message.

In order to start a sequential trace the user must explicitly set the trace token in the process that will send the first message in a sequence.

The trace token of a process is set each time the process matches a message in a receive statement, according to the trace token carried by the received message, empty or not.

On each Erlang node a process can be set as the *system tracer*. This process will receive trace messages each time a message with a trace token is sent or received (if the trace token flag send or 'receive' is set). The system tracer can then print each trace event, write it to a file or whatever suitable.

Note:

The system tracer will only receive those trace events that occur locally within the Erlang node. To get the whole picture of a sequential trace that involves processes on several Erlang nodes, the output from the system tracer on each involved node must be merged (off line).

In the following sections Sequential Tracing and its most fundamental concepts are described.

Trace Token

Each process has a current trace token. Initially the token is empty. When the process sends a message to another process, a copy of the current token will be sent “invisibly” along with the message. The current token of a process is set in two ways, either

1. explicitly by the process itself, through a call to `seq_trace:set_token`, or
2. when a message is received.

In both cases the current token will be set. In particular, if the token of a message received is empty, the current token of the process is set to empty.

A trace token contains a label, and a set of flags. Both the label and the flags are set in 1 and 2 above.

Serial

The trace token contains a component which we call the `Serial` which consists of two integers `Previous` and `Current`. The purpose of `Serial` is uniquely identify each traced event within a trace sequence and to order the messages chronologically and in the different branches if any.

The algorithm for updating `Serial` can be described as follows:

Let each process have two counters `prev_cnt` and `curr_cnt` which both are set to 0 when a process is created. The counters are updated at the following occasions:

- *When the process is about to send a message and the trace token is not empty.*
 Let the `Serial` of the trace token be `tprev` and `tcurr`.

```
curr_cnt := curr_cnt + 1
tprev := prev_cnt
tcurr := curr_cnt
```

 The trace token with `tprev` and `tcurr` is then passed along with the message.
- *When the process calls `seq_trace:print(Label, Info)`, the `Label` matches the label part of the trace token and the trace token print flag is true.*
 The same algorithm as for send above.
- *When a message is received and contains a nonempty trace token.*
 The process trace token is set to the trace token from the message.
 Let the `Serial` of the trace token be `tprev` and `tcurr`.

```
if (curr_cnt < tcurr )
    curr_cnt := tcurr
prev_cnt := tprev
```

The `curr_cnt` of a process is incremented each time the process is involved in a sequential trace. The counter can reach its limit (27 bits) if a process is very long-lived and is involved in much sequential tracing. If the counter overflows it will not be possible to use the `Serial` for ordering of the trace events. To prevent the counter from overflowing in the middle of a sequential trace the function `seq_trace:reset_trace/0` can be called to reset the `prev_cnt` and `curr_cnt` of all processes in the Erlang node. This function will also set all trace tokens in processes and their message queues to empty and will thus stop all ongoing sequential tracing.

Performance considerations

The performance degradation for a system which is enabled for Sequential tracing is negligible as long as no tracing is activated. When tracing is activated there will of course be an extra cost for each traced message but all other messages will be unaffected.

Ports

Sequential tracing is not performed across ports.

If the user for some reason wants to pass the trace token to a port this has to be done manually in the code of the port controlling process. The port controlling processes have to check the appropriate sequential trace settings (as obtained from `seq_trace:get_token/1` and include trace information in the message data sent to their respective ports.

Similarly, for messages received from a port, a port controller has to retrieve trace specific information, and set appropriate sequential trace flags through calls to `seq_trace:set_token/2`.

Distribution

Sequential tracing between nodes is performed transparently. This applies to C-nodes built with `Erl_interface` too. A C-node built with `Erl_interface` only maintains one trace token which means that the C-node will appear as one process from the sequential tracing point of view.

In order to be able to perform sequential tracing between distributed Erlang nodes, the distribution protocol has been extended (in a backward compatible way). An Erlang node which supports sequential tracing can communicate with an older (OTP R3B) node but messages passed within that node can of course not be traced.

Example of Usage

The example shown here will give rough idea of how the new primitives can be used and what kind of output it will produce.

Assume that we have an initiating process with `Pid = <0.30.0>` like this:

```
-module(seqex).
-compile(export_all).

loop(Port) ->
    receive
        {Port,Message} ->
            seq_trace:set_token(label,17),
            seq_trace:set_token('receive',true),
            seq_trace:set_token(print,true),
            seq_trace:print(17,"**** Trace Started ****"),
            call_server ! {self(),the_message};
        {ack,Ack} ->
            ok
```



```
end,
loop(Port).
```

And a registered process 'call_server' with Pid = <0.31.0> like this:

```
loop() ->
  receive
    {PortController,Message} ->
      Ack = {received, Message},
      seq_trace:print(17,"We are here now"),
      PortController ! {ack,Ack}
  end,
  loop().
```

A possible output from the system's sequential_tracer (inspired by AXE-10 and MD-110) could look like:

```
17:<0.30.0> Info {0,1} WITH
"**** Trace Started ****"
17:<0.31.0> Received {0,2} FROM <0.30.0> WITH
{<0.30.0>,the_message}
17:<0.31.0> Info {2,3} WITH
"We are here now"
17:<0.30.0> Received {2,4} FROM <0.31.0> WITH
{ack,{received,the_message}}
```

The implementation of a system tracer process that produces the printout above could look like this:

```
tracer() ->
  receive
    {seq_trace,Label,TraceInfo} ->
      print_trace(Label,TraceInfo,false);
    {seq_trace,Label,TraceInfo,Ts} ->
      print_trace(Label,TraceInfo,Ts);
    Other -> ignore
  end,
  tracer().

print_trace(Label,TraceInfo,false) ->
  io:format("~p:",[Label]),
  print_trace(TraceInfo);
print_trace(Label,TraceInfo,Ts) ->
  io:format("~p ~p:",[Label,Ts]),
  print_trace(TraceInfo).

print_trace({print,Serial,From,_,Info}) ->
  io:format("~p Info ~p WITH~n~p~n", [From,Serial,Info]);
print_trace({'receive',Serial,From,To,Message}) ->
  io:format("~p Received ~p FROM ~p WITH~n~p~n",
    [To,Serial,From,Message]);
print_trace({send,Serial,From,To,Message}) ->
  io:format("~p Sent ~p TO ~p WITH~n~p~n",
    [From,Serial,To,Message]).
```

The code that creates a process that runs the tracer function above and sets that process as the system tracer could look like this:

```
start() ->
    Pid = spawn(?MODULE,tracer,[]),
    seq_trace:set_system_tracer(Pid), % set Pid as the system tracer
    ok.
```

With a function like `test/0` below the whole example can be started.

```
test() ->
    P = spawn(?MODULE, loop, [port]),
    register(call_server, spawn(?MODULE, loop, [])),
    start(),
    P ! {port,message}.
```

user

Erlang Module

`user` is a server which responds to all the messages defined in the I/O interface. The code in `user.erl` can be used as a model for building alternative I/O servers.

Exports

`start() -> void()`

Starts the basic standard I/O server for the user interface port.

wrap_log_reader

Erlang Module

`wrap_log_reader` is a function to read internally formatted wrap disk logs, refer to `disk_log(3)`. `wrap_log_reader` does not interfere with `disk_log` activities; there is however a known bug in this version of the `wrap_log_reader`, see chapter bugs below.

A wrap disk log file consists of several files, called index files. A log file can be opened and closed. It is also possible to open just one index file separately. If a 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 chunk 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)`

app

File

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

This file is read by the application controller when an application is loaded. It is also used by the functions in `systools` when generating start scripts etc.

FILE SYNTAX

The application resource file should be called `Application.app` where `Application` is the name of the application. The file should be located in the `ebin` directory for the application.

The `.app` file contains one single Erlang term, which is called an *application specification*. The file has the following syntax:

```
{application, Application,
  [{description, Description},
   {vsn, VsN},
   {id, Id},
   {modules, [Module1, .., ModuleN]}},
  {maxT, MaxT},
  {registered, [Name1, .., NameN]},
  {applications, [App1, .., AppN]},
  {included_applications, [App1, .., AppN]},
  {env, [{Par1, Val1}, .., {ParN, ValN}]},
  {mod, {Module, StartArgs}},
  {start_phases, [{Phase1, PhaseArgs1}, .., {PhaseN, PhaseArgsN}]}}.
```

`Application = atom()` is the name of the application.

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

The functions in `systools` require more information. If they are used, the following keys are mandatory: `description`, `vsn`, `modules`, `registered` and `applications`. The other keys are ignored by `systools`.

- `{description, Description}`
`Description = string()` is a textual description of the application. Defaults to the empty string.
- `{vsn, VsN}`
`VsN = string()` is the version of the application. Defaults to the empty string.

- `{id, Id}`
`Id = string()` is the product identification of the application. Defaults to the empty string.
- `{modules, Modules}`
`Modules = [atom()]` is a list of all the modules introduced by this application. `systools` uses this list when generating start scripts and tar files. A module can only be defined in one application. Defaults to the empty list.
It is also allowed to list a module as `{Module, Vsn}`, where `Vsn = term()`. This has no practical implication, however, as there is no check that `Vsn` is the same value as the `vs_n` attribute of the module. The format is retained for backwards compatibility only.
- `{maxT, MaxT}`
`MaxT = int() | infinity` is the maximum time in milliseconds that the application is allowed to run. After the specified time the application will automatically terminate. Defaults to `infinity`.
- `{registered, Registered}`
`Registered = [atom()]` is a list of all the names of registered processes started in this application. `systools` uses this list to detect name clashes between different applications. Defaults to the empty list.
- `{applications, Applications}`
`Applications = [atom()]` is a list of all the names of applications which must be started before this application is started. `systools` uses this list to generate correct start scripts. Defaults to the empty list, but note that all applications implemented as supervision trees have dependencies to (at least) `kernel` and `stdlib`.
- `{included_applications, Applications}`
`Applications = [atom()]` is a list of all the names of applications which are included by this application. When this application is started, all included application will automatically be loaded, but not started, by the application controller. Processes implemented in an included application should be placed underneath a supervisor in the primary application. Defaults to the empty list.
- `{env, Env}`
`Env = [{Par, Val}]`, where `Par = atom()` and `Val = term()`, is a list of configuration parameters used by the application. The value of a configuration parameter is retrieved by calling `application:get_env/1,2`. The values in the application resource file can be overridden by values in a configuration file (see `config(4)`) or by command line flags (see `erl(1)`). Defaults to the empty list.
- `{mod, Mod}`
`Mod = {Module, StartArgs}`, where `Module = atom()` and `StartArgs = term()`, specifies the application callback module and the start argument, see `application(3)`.
The `mod` key is mandatory for applications implemented as supervision trees, but should be omitted for applications which are code libraries, such as the application `STDLIB`.
- `{start_phases, StartPhases}`
`StartPhases = [{Phase, PhaseArgs}]`, where `Phase = atom()` and `PhaseArgs = [term()]`, is a list of start phases and the attached start arguments for the application. After starting the application, the application master will evaluate the function `Mod:start_phase(Phase, Type, PhaseArgs)` for each defined start phase, where `Mod` is the callback module as defined by the `mod` key. This extended start procedure is intended for synchronized startup of included applications, refer to

the chapter about *OTP Design Principles* for more information. Defaults to `undefined`.

The following key is no longer used but still allowed (and ignored) to retain backwards compatibility.

- `{maxP,N}`
N = `int()` was the maximum number of processes the application was allowed to start. Defaults to `infinity`.

SEE ALSO

`application(3)`, `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.

SEE ALSO

`application(3)`, `systools(3)`

Index of Modules and Functions

Modules are typed in *this* way.
Functions are typed in *this* way.

abcast/2
 rpc , 172

abcast/3
 rpc , 172

abs/1
 erlang , 73

accept/1
 gen_tcp , 134

accept/2
 gen_tcp , 134

accessible_logs/0
 disk_log , 52

add_path/1
 code , 44

add_patha/1
 code , 44

add_paths/1
 code , 44

add_pathsa/1
 code , 45

add_pathsz/1
 code , 44

add_pathz/1
 code , 44

add_report_handler/1
 error_logger , 114

add_report_handler/2
 error_logger , 114

add_slave/1
 erl_boot_server , 64

all_loaded/0
 code , 47

allow/1

net_kernel , 165

alog/2
 disk_log , 52

alog_terms/2
 disk_log , 52

application
 get_all_env/0, 32
 get_all_env/1, 32
 get_all_key/0, 32
 get_all_key/1, 32
 get_application/0, 32
 get_application/1, 32
 get_env/1, 32
 get_env/2, 32
 get_key/1, 33
 get_key/2, 33
 load/1, 33
 load/2, 33
 loaded_applications/0, 34
 Module:config_change/3, 37
 Module:prep_stop/1, 38
 Module:start/2, 37
 Module:start_phase/3, 38
 Module:stop/1, 39
 permit/2, 34
 set_env/3, 35
 start/1, 35
 start/2, 35
 start_type/0, 35
 stop/1, 36
 takeover/2, 36
 unload/1, 36
 which_applications/0, 37

apply/3
 erlang , 73

async_call/4
 rpc , 174

atom_to_list/1

- erlang* , 73
- auth*
 - cookie*/0, 41
 - cookie*/1, 42
 - exists*/1, 41
 - is_auth*/1, 41
 - node_cookie*/2, 41, 42
 - open*/1, 41
 - start*/0, 41
 - stop*/0, 41
- balog*/2
 - disk_log* , 52
- balog_terms*/2
 - disk_log* , 52
- binary_to_list*/1
 - erlang* , 74
- binary_to_list*/3
 - erlang* , 74
- binary_to_term*/1
 - erlang* , 74
- block*/1
 - disk_log* , 53
- block*/2
 - disk_log* , 53
- block_call*/4
 - rpc* , 171
- blog*/2
 - disk_log* , 58
- blog_terms*/2
 - disk_log* , 58
- boot*/1
 - init* , 157
- breopen*/3
 - disk_log* , 62
- btruncate*/2
 - disk_log* , 63
- call*/4
 - rpc* , 171
- cast*/4
 - rpc* , 171
- change_group*/2
 - file* , 116
- change_header*/2
 - disk_log* , 53
- change_notify*/3
 - disk_log* , 53
- change_owner*/2
 - file* , 116
- change_owner*/3
 - file* , 116
- change_size*/2
 - disk_log* , 53
- change_time*/2
 - file* , 116
- change_time*/3
 - file* , 116
- chunk*/1
 - wrap_log_reader* , 184
- chunk*/2
 - disk_log* , 54
 - wrap_log_reader* , 184
- chunk*/3
 - disk_log* , 54
- chunk_info*/1
 - disk_log* , 55
- chunk_step*/3
 - disk_log* , 55
- clash*/0
 - code* , 49
- clear_cmd*/0
 - heart* , 149
- close*/1
 - disk_log* , 55
 - file* , 116
 - gen_tcp* , 134
 - gen_udp* , 137
 - inet* , 151
 - wrap_log_reader* , 185
- cmd*/1
 - os* , 166
- code*
 - add_path*/1, 44
 - add_patha*/1, 44
 - add_paths*/1, 44
 - add_pathsa*/1, 45
 - add_pathsz*/1, 44
 - add_pathz*/1, 44
 - all_loaded*/0, 47

- clash/0, 49
- compiler_dir/0, 48
- del_path/1, 45
- delete/1, 46
- ensure_loaded/1, 46
- get_object_code/1, 48
- get_path/0, 44
- is_loaded/1, 46
- lib_dir/0, 47
- lib_dir/1, 47
- load_abs/1, 45
- load_binary/3, 47
- load_file/1, 45
- objfile_extension/0, 48
- priv_dir/1, 48
- purge/1, 46
- replace_path/2, 45
- root_dir/0, 47
- set_path/1, 44
- soft_purge/1, 46
- start/0, 43
- start/1, 43
- start_link/0, 43
- start_link/1, 43
- stick_dir/1, 48
- stop/0, 47
- unstick_dir/1, 49
- which/1, 49
- compiler_dir/0
 - code*, 48
- concat_binary/1
 - erlang*, 75
- connect/3
 - gen_tcp*, 134
- connect/4
 - gen_tcp*, 134
- connect_node/1
 - net_kernel*, 165
- consult/1
 - file*, 117
- controlling_process/2
 - gen_tcp*, 135
 - gen_udp*, 137
- cookie/0
 - auth*, 41
- cookie/1
 - auth*, 42
- copy/2
 - file*, 117
- copy/3
 - file*, 117
- create/1
 - pg2*, 169
- date/0
 - erlang*, 75
- del_dir/1
 - file*, 117
- del_lock/1
 - global*, 140
- del_lock/2
 - global*, 140
- del_path/1
 - code*, 45
- delete/1
 - code*, 46
 - file*, 118
 - pg2*, 169
- delete_report_handler/1
 - error_logger*, 114
- delete_slave/1
 - erl_boot_server*, 64
- disk_log*
 - accessible_logs*/0, 52
 - alog*/2, 52
 - alog_terms*/2, 52
 - balog*/2, 52
 - balog_terms*/2, 52
 - block*/1, 53
 - block*/2, 53
 - blog*/2, 58
 - blog_terms*/2, 58
 - breopen*/3, 62
 - btruncate*/2, 63
 - change_header*/2, 53
 - change_notify*/3, 53
 - change_size*/2, 53
 - chunk*/2, 54
 - chunk*/3, 54
 - chunk_info*/1, 55
 - chunk_step*/3, 55
 - close*/1, 55
 - format_error*/1, 56
 - inc_wrap_file*/1, 56
 - info*/1, 56
 - lclose*/1, 57

lclose/2, 57	date/0, 75
log/2, 58	element/2, 76
log_terms/2, 58	erase/0, 76
open/1, 59	erase/1, 76
pid2name/1, 62	erlang:append_element/2, 73
reopen/2, 62	erlang:binary_to_float/1, 73
reopen/3, 62	erlang:bump_reductions/1, 74
sync/1, 62	erlang:cancel_timer/1, 74
truncate/1, 63	erlang:check_process_code/2, 75
truncate/2, 63	erlang:delete_module/1, 75
unblock/1, 63	erlang:demonitor/1, 76
dns_hostname/1	erlang:disconnect_node/1, 76
<i>net_adm</i> , 162	erlang:display/1, 76
element/2	erlang:fault/1, 77
<i>erlang</i> , 76	erlang:fault/2, 77
ensure_loaded/1	erlang:float_to_binary/2, 78
<i>code</i> , 46	erlang:fun_info/1, 78
erase/0	erlang:fun_info/2, 78
<i>erlang</i> , 76	erlang:fun_to_list/1, 79
erase/1	erlang:function_exported/3, 79
<i>erlang</i> , 76	erlang:garbage_collect/0, 79
<i>erl_boot_server</i>	erlang:garbage_collect/1, 79
add_slave/1, 64	erlang:get_cookie/0, 79
delete_slave/1, 64	erlang:hash/2, 80
start/1, 64	erlang:info/1, 81
start_link/1, 64	erlang:is_builtin/3, 81
which_slaves/0, 65	erlang:load_module/2, 83
<i>erl_ddll</i>	erlang:loaded/0, 83
format_error/1, 67	erlang:localtime/0, 83
load_driver/2, 66	erlang:localtime_to_universaltime/1, 83
loaded_drivers/0, 67	erlang:make_tuple/2, 84
start/0, 66	erlang:md5/1, 84
start_link/0, 66	erlang:md5_final/1, 84
stop/0, 66	erlang:md5_init/0, 84
unload_driver/1, 66	erlang:md5_update/2, 84
<i>erl_prim_loader</i>	erlang:module_loaded/1, 84
get_file/1, 69	erlang:monitor/2, 85
get_path/0, 70	erlang:phash/2, 89
set_path/1, 70	erlang:port_info/2, 91
start/3, 69	erlang:port_to_list/1, 91
<i>erlang</i>	erlang:ports/0, 92
abs/1, 73	erlang:pre_loaded/0, 92
apply/3, 73	erlang:process_display/2, 92
atom_to_list/1, 73	erlang:purge_module/1, 95
binary_to_list/1, 74	erlang:read_timer/1, 95
binary_to_list/3, 74	erlang:ref_to_list/1, 95
binary_to_term/1, 74	erlang:resume_process/1, 96
concat_binary/1, 75	erlang:send_after/3, 96
	erlang:set_cookie/2, 97
	erlang:start_timer/3, 100
	erlang:suspend_process/1, 101
	erlang:system_flag/2, 101

erlang:system_info/1, 101	registered/0, 96
erlang:trace/3, 103	round/1, 96
erlang:trace_info/2, 105	self/0, 96
erlang:trace_pattern/2, 106	setelement/3, 97
erlang:trace_pattern/3, 106	size/1, 97
erlang:universaltime/0, 108	spawn/1, 97
erlang:universaltime_to_localtime/1, 108	spawn/2, 97
	spawn/3, 97
erlang:yield/0, 108	spawn/4, 98
exit/1, 76	spawn_link/1, 98
exit/2, 77	spawn_link/2, 98
float/1, 77	spawn_link/3, 98
float_to_list/1, 78	spawn_link/4, 98
get/0, 79	spawn_opt/4, 99
get/1, 79	split_binary/2, 99
get_keys/1, 79	statistics/1, 100
group_leader/0, 80	term_to_binary/1, 102
group_leader/2, 80	term_to_binary/2, 102
halt/0, 80	throw/1, 102
halt/1, 80	time/0, 102
hd/1, 81	tl/1, 103
integer_to_list/1, 81	trunc/1, 107
is_alive/0, 81	tuple_to_list/1, 107
is_process_alive/1, 81	unlink/1, 108
length/1, 81	unregister/1, 108
link/1, 81	whereis/1, 108
list_to_atom/1, 82	
list_to_binary/1, 82	erlang:append_element/2
list_to_float/1, 82	<i>erlang</i> , 73
list_to_integer/1, 82	erlang:binary_to_float/1
list_to_pid/1, 82	<i>erlang</i> , 73
list_to_tuple/1, 82	erlang:bump_reductions/1
make_ref/0, 84	<i>erlang</i> , 74
monitor_node/2, 85	erlang:cancel_timer/1
node/0, 86	<i>erlang</i> , 74
node/1, 86	erlang:check_process_code/2
nodes/0, 86	<i>erlang</i> , 75
nodes/1, 86	erlang:delete_module/1
now/0, 87	<i>erlang</i> , 75
open_port/2, 87	erlang:demonitor/1
pid_to_list/1, 89	<i>erlang</i> , 76
port_call/3, 91	erlang:disconnect_node/1
port_close/1, 89	<i>erlang</i> , 76
port_command/2, 90	erlang:display/1
port_connect/2, 90	<i>erlang</i> , 76
port_control/3, 90	erlang:fault/1
process_flag/2, 92	<i>erlang</i> , 77
process_flag/3, 93	erlang:fault/2
process_info/1, 93	<i>erlang</i> , 77
process_info/2, 94	
processes/0, 95	
put/2, 95	
register/2, 96	

erlang:float_to_binary/2	<i>erlang</i> , 78	<i>erlang</i> , 85
erlang:fun_info/1	<i>erlang</i> , 78	erlang:phash/2
erlang:fun_info/2	<i>erlang</i> , 78	<i>erlang</i> , 89
erlang:fun_to_list/1	<i>erlang</i> , 79	erlang:port_info/2
erlang:function_exported/3	<i>erlang</i> , 79	<i>erlang</i> , 91
erlang:garbage_collect/0	<i>erlang</i> , 79	erlang:port_to_list/1
erlang:garbage_collect/1	<i>erlang</i> , 79	<i>erlang</i> , 91
erlang:get_cookie/0	<i>erlang</i> , 79	erlang:ports/0
erlang:hash/2	<i>erlang</i> , 80	<i>erlang</i> , 92
erlang:info/1	<i>erlang</i> , 81	erlang:pre_loaded/0
erlang:is_builtin/3	<i>erlang</i> , 81	<i>erlang</i> , 92
erlang:load_module/2	<i>erlang</i> , 83	erlang:process_display/2
erlang:loaded/0	<i>erlang</i> , 83	<i>erlang</i> , 92
erlang:localtime/0	<i>erlang</i> , 83	erlang:purge_module/1
erlang:localtime_to_universaltime/1	<i>erlang</i> , 83	<i>erlang</i> , 95
erlang:make_tuple/2	<i>erlang</i> , 84	erlang:read_timer/1
erlang:md5/1	<i>erlang</i> , 84	<i>erlang</i> , 95
erlang:md5_final/1	<i>erlang</i> , 84	erlang:ref_to_list/1
erlang:md5_init/0	<i>erlang</i> , 84	<i>erlang</i> , 95
erlang:md5_update/2	<i>erlang</i> , 84	erlang:resume_process/1
erlang:module_loaded/1	<i>erlang</i> , 84	<i>erlang</i> , 96
erlang:monitor/2		erlang:send_after/3
		<i>erlang</i> , 96
		erlang:set_cookie/2
		<i>erlang</i> , 97
		erlang:start_timer/3
		<i>erlang</i> , 100
		erlang:suspend_process/1
		<i>erlang</i> , 101
		erlang:system_flag/2
		<i>erlang</i> , 101
		erlang:system_info/1
		<i>erlang</i> , 101
		erlang:trace/3
		<i>erlang</i> , 103
		erlang:trace_info/2
		<i>erlang</i> , 105
		erlang:trace_pattern/2
		<i>erlang</i> , 106
		erlang:trace_pattern/3
		<i>erlang</i> , 106
		erlang:universaltime/0
		<i>erlang</i> , 108

- erlang:universaltime_to_localtime/1
 - erlang* , 108
- erlang:yield/0
 - erlang* , 108
- error_handler*
 - undefined_function/3, 109
 - undefined_lambda/3, 109
- error_logger*
 - add_report_handler/1, 114
 - add_report_handler/2, 114
 - delete_report_handler/1, 114
 - error_msg/1, 113
 - error_msg/2, 113
 - error_report/1, 111
 - error_report/2, 112
 - format/2, 113
 - info_msg/1, 113
 - info_msg/2, 113
 - info_report/1, 112
 - info_report/2, 112
 - logfile/1, 113
 - start/0, 111
 - start_link/0, 111
 - swap_handler/1, 114
 - tty/1, 113
- error_msg/1
 - error_logger* , 113
- error_msg/2
 - error_logger* , 113
- error_report/1
 - error_logger* , 111
- error_report/2
 - error_logger* , 112
- eval/1
 - file* , 118
- eval_everywhere/3
 - rpc* , 172
- eval_everywhere/4
 - rpc* , 172
- exists/1
 - auth* , 41
- exit/1
 - erlang* , 76
- exit/2
 - erlang* , 77
- file*
 - change_group/2, 116
 - change_owner/2, 116
 - change_owner/3, 116
 - change_time/2, 116
 - change_time/3, 116
 - close/1, 116
 - consult/1, 117
 - copy/2, 117
 - copy/3, 117
 - del_dir/1, 117
 - delete/1, 118
 - eval/1, 118
 - file_info/1, 118
 - format_error/1, 119
 - get_cwd/0, 119
 - get_cwd/1, 119
 - ipread_s32bu_p32bu/3, 119
 - list_dir/1, 120
 - make_dir/1, 120
 - make_link/2, 120
 - make_symlink/2, 120
 - open/2, 121
 - path_consult/2, 122
 - path_eval/2, 123
 - path_open/3, 123
 - position/2, 123
 - pread/3, 124
 - pread/4, 124
 - pwrite/3, 124
 - pwrite/4, 124
 - read/2, 124
 - read_file/1, 124
 - read_file_info/1, 125
 - read_link/1, 126
 - read_link_info/1, 126
 - rename/2, 126
 - set_cwd/1, 127
 - sync/1, 127
 - truncate/1, 127
 - write/2, 127
 - write_file/2, 127
 - write_file_info/2, 128
- file_info/1
 - file* , 118
- find_executable/1
 - os* , 166
- find_executable/2
 - os* , 166
- float/1
 - erlang* , 77

float_to_list/1 <i>erlang</i> , 78	get_argument/1 <i>init</i> , 158
format/2 <i>error_logger</i> , 113	get_arguments/0 <i>init</i> , 157
format_error/1 <i>disk_log</i> , 56 <i>erl_ddll</i> , 67 <i>file</i> , 119 <i>inet</i> , 150	get_closest_pid/1 <i>pg2</i> , 169
<i>gen_tcp</i> accept/1, 134 accept/2, 134 close/1, 134 connect/3, 134 connect/4, 134 controlling_process/2, 135 listen/2, 135 recv/2, 135 recv/3, 135 send/2, 136	get_cwd/0 <i>file</i> , 119
<i>gen_udp</i> close/1, 137 controlling_process/2, 137 open/1, 137 open/2, 137 recv/2, 138 recv/3, 138 send/4, 138	get_cwd/1 <i>file</i> , 119
get/0 <i>erlang</i> , 79	get_env/1 <i>application</i> , 32
get/1 <i>erlang</i> , 79	get_env/2 <i>application</i> , 32
get_all_env/0 <i>application</i> , 32	get_file/1 <i>erl_prim_loader</i> , 69
get_all_env/1 <i>application</i> , 32	get_key/1 <i>application</i> , 33
get_all_key/0 <i>application</i> , 32	get_key/2 <i>application</i> , 33
get_all_key/1 <i>application</i> , 32	get_keys/1 <i>erlang</i> , 79
get_application/0 <i>application</i> , 32	get_local_members/1 <i>pg2</i> , 170
get_application/1 <i>application</i> , 32	get_members/1 <i>pg2</i> , 170
get_args/0 <i>init</i> , 158	get_object_code/1 <i>code</i> , 48
	get_path/0 <i>code</i> , 44 <i>erl_prim_loader</i> , 70
	get_plain_arguments/0 <i>init</i> , 158
	get_status/0 <i>init</i> , 159
	get_system_tracer/0 <i>seq_trace</i> , 177
	get_token/0 <i>seq_trace</i> , 176
	get_token/1 <i>seq_trace</i> , 176
	getaddr/2 <i>inet</i> , 152

getenv/0	sync/0, 146
os, 166	whereis_name/1, 147
getenv/1	whereis_name/3, 147
os, 167	global_groups/0
gethostbyaddr/1	global_group, 145
inet, 150	group_leader/0
gethostbyname/1	erlang, 80
inet, 150	group_leader/2
gethostbyname/2	erlang, 80
inet, 151	
gethostname/0	halt/0
inet, 151	erlang, 80
getpid/0	halt/1
os, 167	erlang, 80
global	hd/1
del_lock/1, 140	erlang, 81
del_lock/2, 140	heart
notify_all_name/3, 140	clear_cmd/0, 149
random_exit_name/3, 140	set_cmd/1, 149
random_notify_name/3, 140	start/0, 148
re_register_name/2, 141	host_file/0
re_register_name/3, 141	net_adm, 162
register_name/2, 140	
register_name/3, 140	inc_wrap_file/1
registered_names/0, 141	disk_log, 56
send/2, 141	
set_lock/1, 141	inet
set_lock/2, 141	close/1, 151
set_lock/3, 141	format_error/1, 150
start/0, 142	getaddr/2, 152
start_link/0, 142	gethostbyaddr/1, 150
stop/0, 142	gethostbyname/1, 150
sync/0, 142	gethostbyname/2, 151
trans/2, 142	gethostname/0, 151
trans/3, 142	peername/1, 151
trans/4, 142	port/1, 151
unregister_name/1, 143	setopts/2, 152
whereis_name/1, 143	sockname/1, 151
global_group	
global_groups/0, 145	info/0
info/0, 145	global_group, 145
monitor_nodes/1, 146	info/1
own_nodes/0, 146	disk_log, 56
registered_names/2, 146	
send/2, 146	info_msg/1
send/4, 146	error_logger, 113
start/0, 147	
start_link/0, 147	info_msg/2
stop/0, 147	error_logger, 113
	info_report/1

<i>error_logger</i> , 112	<i>erlang</i> , 81
info_report/2	list_dir/1
<i>error_logger</i> , 112	<i>file</i> , 120
init	list_to_atom/1
boot/1, 157	<i>erlang</i> , 82
get_args/0, 158	list_to_binary/1
get_argument/1, 158	<i>erlang</i> , 82
get_arguments/0, 157	list_to_float/1
get_plain_arguments/0, 158	<i>erlang</i> , 82
get_status/0, 159	list_to_integer/1
reboot/0, 158	<i>erlang</i> , 82
restart/0, 158	list_to_pid/1
script_id/0, 159	<i>erlang</i> , 82
stop/0, 158	list_to_tuple/1
integer_to_list/1	<i>erlang</i> , 82
<i>erlang</i> , 81	listen/2
ipread_s32bu_p32bu/3	<i>gen_tcp</i> , 135
<i>file</i> , 119	load/1
is_alive/0	<i>application</i> , 33
<i>erlang</i> , 81	load/2
is_auth/1	<i>application</i> , 33
<i>auth</i> , 41	load_abs/1
is_loaded/1	<i>code</i> , 45
<i>code</i> , 46	load_binary/3
is_process_alive/1	<i>code</i> , 47
<i>erlang</i> , 81	load_driver/2
join/2	<i>erl_ddll</i> , 66
<i>pg2</i> , 170	load_file/1
kernel_apply/3	<i>code</i> , 45
<i>net_kernel</i> , 164	loaded_applications/0
lclose/1	<i>application</i> , 34
<i>disk_log</i> , 57	loaded_drivers/0
lclose/2	<i>erl_ddll</i> , 67
<i>disk_log</i> , 57	localhost/0
leave/2	<i>net_adm</i> , 162
<i>pg2</i> , 170	log/2
length/1	<i>disk_log</i> , 58
<i>erlang</i> , 81	log_terms/2
lib_dir/0	<i>disk_log</i> , 58
<i>code</i> , 47	logfile/1
lib_dir/1	<i>error_logger</i> , 113
<i>code</i> , 47	make_dir/1
link/1	

- file* , 120
- make_link/2
 - file* , 120
- make_ref/0
 - erlang* , 84
- make_symlink/2
 - file* , 120
- Module:config_change/3
 - application* , 37
- Module:prep_stop/1
 - application* , 38
- Module:start/2
 - application* , 37
- Module:start_phase/3
 - application* , 38
- Module:stop/1
 - application* , 39
- monitor_node/2
 - erlang* , 85
- monitor_nodes/1
 - global_group* , 146
 - net_kernel* , 164
- multi_server_call/2
 - rpc* , 173
- multi_server_call/3
 - rpc* , 173
- multicall/3
 - rpc* , 172
- multicall/4
 - rpc* , 173
- names/0
 - net_adm* , 162
- nb_yield/1
 - rpc* , 174
- nb_yield/2
 - rpc* , 174
- net_adm*
 - dns_hostname/1, 162
 - host_file/0, 162
 - localhost/0, 162
 - names/0, 162
 - ping/1, 162
 - world/0, 162
 - world_list/2, 162

- net_kernel*
 - allow/1, 165
 - connect_node/1, 165
 - kernel_apply/3, 164
 - monitor_nodes/1, 164
- node/0
 - erlang* , 86
- node/1
 - erlang* , 86
- node_cookie/2
 - auth* , 41, 42
- nodes/0
 - erlang* , 86
- nodes/1
 - erlang* , 86
- notify_all_name/3
 - global* , 140
- now/0
 - erlang* , 87
- objfile_extension/0
 - code* , 48
- open/1
 - auth* , 41
 - disk_log* , 59
 - gen_udp* , 137
 - wrap_log_reader* , 185
- open/2
 - file* , 121
 - gen_udp* , 137
 - wrap_log_reader* , 185
- open_port/2
 - erlang* , 87
- os*
 - cmd/1, 166
 - find_executable/1, 166
 - find_executable/2, 166
 - getenv/0, 166
 - getenv/1, 167
 - getpid/0, 167
 - putenv/2, 167
 - type/0, 167
 - version/0, 167
- own_nodes/0
 - global_group* , 146
- parallel_eval/1

- rpc* , 174
- path_consult/2
 - file* , 122
- path_eval/2
 - file* , 123
- path_open/3
 - file* , 123
- peername/1
 - inet* , 151
- permit/2
 - application* , 34
- pg2
 - create/1, 169
 - delete/1, 169
 - get_closest_pid/1, 169
 - get_local_members/1, 170
 - get_members/1, 170
 - join/2, 170
 - leave/2, 170
 - start/0, 170
 - start_link/0, 170
 - which_groups/0, 170
- pid2name/1
 - disk_log* , 62
- pid_to_list/1
 - erlang* , 89
- pinfo/1
 - rpc* , 174
- pinfo/2
 - rpc* , 174
- ping/1
 - net_adm* , 162
- pmap/4
 - rpc* , 174
- port/1
 - inet* , 151
- port_call/3
 - erlang* , 91
- port_close/1
 - erlang* , 89
- port_command/2
 - erlang* , 90
- port_connect/2
 - erlang* , 90
- port_control/3
 - erlang* , 90
- position/2
 - file* , 123
- pread/3
 - file* , 124
- pread/4
 - file* , 124
- print/1
 - seq_trace* , 176
- print/2
 - seq_trace* , 177
- priv_dir/1
 - code* , 48
- process_flag/2
 - erlang* , 92
- process_flag/3
 - erlang* , 93
- process_info/1
 - erlang* , 93
- process_info/2
 - erlang* , 94
- processes/0
 - erlang* , 95
- purge/1
 - code* , 46
- put/2
 - erlang* , 95
- putenv/2
 - os* , 167
- pwrite/3
 - file* , 124
- pwrite/4
 - file* , 124
- random_exit_name/3
 - global* , 140
- random_notify_name/3
 - global* , 140
- re_register_name/2
 - global* , 141
- re_register_name/3
 - global* , 141

read/2	<i>code</i> , 47
<i>file</i> , 124	
read_file/1	round/1
<i>file</i> , 124	<i>erlang</i> , 96
read_file_info/1	<i>rpc</i>
<i>file</i> , 125	abcast/2, 172
read_link/1	abcast/3, 172
<i>file</i> , 126	async_call/4, 174
read_link_info/1	block_call/4, 171
<i>file</i> , 126	call/4, 171
reboot/0	cast/4, 171
<i>init</i> , 158	eval_everywhere/3, 172
recv/2	eval_everywhere/4, 172
<i>gen_tcp</i> , 135	multi_server_call/2, 173
<i>gen_udp</i> , 138	multi_server_call/3, 173
recv/3	multicall/3, 172
<i>gen_tcp</i> , 135	multicall/4, 173
<i>gen_udp</i> , 138	nb_yield/1, 174
register/2	nb_yield/2, 174
<i>erlang</i> , 96	parallel_eval/1, 174
register_name/2	pinfo/1, 174
<i>global</i> , 140	pinfo/2, 174
register_name/3	pmap/4, 174
<i>global</i> , 140	safe_multi_server_call/2, 173
registered/0	safe_multi_server_call/3, 174
<i>erlang</i> , 96	sbcast/2, 172
registered_names/0	sbcast/3, 172
<i>global</i> , 141	server_call/4, 172
registered_names/2	start/0, 171
<i>global-group</i> , 146	stop/0, 171
rename/2	yield/1, 174
<i>file</i> , 126	
reopen/2	safe_multi_server_call/2
<i>disk_log</i> , 62	<i>rpc</i> , 173
reopen/3	safe_multi_server_call/3
<i>disk_log</i> , 62	<i>rpc</i> , 174
replace_path/2	sbcast/2
<i>code</i> , 45	<i>rpc</i> , 172
reset_trace/0	sbcast/3
<i>seq_trace</i> , 177	<i>rpc</i> , 172
restart/0	script_id/0
<i>init</i> , 158	<i>init</i> , 159
root_dir/0	self/0
	<i>erlang</i> , 96
	send/2
	<i>gen_tcp</i> , 136
	<i>global</i> , 141
	<i>global-group</i> , 146
	send/4

- gen_udp* , 138
- global_group* , 146
- seq_trace*
 - get_system_tracer*/0, 177
 - get_token*/0, 176
 - get_token*/1, 176
 - print*/1, 176
 - print*/2, 177
 - reset_trace*/0, 177
 - set_system_tracer*/1, 177
 - set_token*/1, 176
 - set_token*/2, 175
- server_call*/4
 - rpc* , 172
- set_cmd*/1
 - heart* , 149
- set_cwd*/1
 - file* , 127
- set_env*/3
 - application* , 35
- set_lock*/1
 - global* , 141
- set_lock*/2
 - global* , 141
- set_lock*/3
 - global* , 141
- set_path*/1
 - code* , 44
 - erl_prim_loader* , 70
- set_system_tracer*/1
 - seq_trace* , 177
- set_token*/1
 - seq_trace* , 176
- set_token*/2
 - seq_trace* , 175
- setelement*/3
 - erlang* , 97
- setopts*/2
 - inet* , 152
- size*/1
 - erlang* , 97
- sockname*/1
 - inet* , 151
- soft_purge*/1
 - code* , 46
- spawn*/1
 - erlang* , 97
- spawn*/2
 - erlang* , 97
- spawn*/3
 - erlang* , 97
- spawn*/4
 - erlang* , 98
- spawn_link*/1
 - erlang* , 98
- spawn_link*/2
 - erlang* , 98
- spawn_link*/3
 - erlang* , 98
- spawn_link*/4
 - erlang* , 98
- spawn_opt*/4
 - erlang* , 99
- split_binary*/2
 - erlang* , 99
- start*/0
 - auth* , 41
 - code* , 43
 - erl_ddll* , 66
 - error_logger* , 111
 - global* , 142
 - global_group* , 147
 - heart* , 148
 - pg2* , 170
 - rpc* , 171
 - user* , 183
- start*/1
 - application* , 35
 - code* , 43
 - erl_boot_server* , 64
- start*/2
 - application* , 35
- start*/3
 - erl_prim_loader* , 69
- start_link*/0
 - code* , 43
 - erl_ddll* , 66
 - error_logger* , 111
 - global* , 142
 - global_group* , 147
 - pg2* , 170

start_link/1	<i>global</i> , 142
<i>code</i> , 43	
<i>erl_boot_server</i> , 64	
start_type/0	trunc/1
<i>application</i> , 35	<i>erlang</i> , 107
statistics/1	truncate/1
<i>erlang</i> , 100	<i>disk_log</i> , 63
stick_dir/1	<i>file</i> , 127
<i>code</i> , 48	truncate/2
stop/0	<i>disk_log</i> , 63
<i>auth</i> , 41	tty/1
<i>code</i> , 47	<i>error_logger</i> , 113
<i>erl_ddll</i> , 66	tuple_to_list/1
<i>global</i> , 142	<i>erlang</i> , 107
<i>global_group</i> , 147	type/0
<i>init</i> , 158	<i>os</i> , 167
<i>rpc</i> , 171	
stop/1	unblock/1
<i>application</i> , 36	<i>disk_log</i> , 63
swap_handler/1	undefined_function/3
<i>error_logger</i> , 114	<i>error_handler</i> , 109
sync/0	undefined_lambda/3
<i>global</i> , 142	<i>error_handler</i> , 109
<i>global_group</i> , 146	unlink/1
sync/1	<i>erlang</i> , 108
<i>disk_log</i> , 62	unload/1
<i>file</i> , 127	<i>application</i> , 36
	unload_driver/1
takeover/2	<i>erl_ddll</i> , 66
<i>application</i> , 36	unregister/1
term_to_binary/1	<i>erlang</i> , 108
<i>erlang</i> , 102	unregister_name/1
term_to_binary/2	<i>global</i> , 143
<i>erlang</i> , 102	unstick_dir/1
throw/1	<i>code</i> , 49
<i>erlang</i> , 102	<i>user</i>
time/0	start/0, 183
<i>erlang</i> , 102	
tl/1	version/0
<i>erlang</i> , 103	<i>os</i> , 167
trans/2	whereis/1
<i>global</i> , 142	<i>erlang</i> , 108
trans/3	whereis_name/1
<i>global</i> , 142	<i>global</i> , 143
trans/4	<i>global_group</i> , 147

whereis_name/3
 global_group , 147

which/1
 code , 49

which_applications/0
 application , 37

which_groups/0
 pg2 , 170

which_slaves/0
 erl_boot_server , 65

world/0
 net_adm , 162

world_list/2
 net_adm , 162

wrap_log_reader
 chunk/1, 184
 chunk/2, 184
 close/1, 185
 open/1, 185
 open/2, 185

write/2
 file , 127

write_file/2
 file , 127

write_file_info/2
 file , 128

yield/1
 rpc , 174